ZOPE 3

DEVELOPER'S HANDBOOK

by Stephan Richter
with input from the Zope 3 community

bookcover design by Marilyn Choo-Ellis (www.texastoast.com)

Contents

Preface

The preface will be a brief introduction into Zope 3 and its capabilities as well as into Python , the programming language Zope is written in.

What is Zope?

What is Zope? While this sounds like a simple question that should be answered in a line or two, I often find myself in situations where I am unable to simple say: "It is an Open-Source Application Server." or "It is a Content Management System.". Both of these descriptions are true, but they are really putting a limit on Zope that simply does not exist. So before I will give my definition of Zope, let's collect some of the solutions Zope has been used for. As mentioned above, many people use Zope as a Content Management System, which are usually Web-based (browser managed) systems. Basically the users can manage the content of a page through a set of Web forms, workflows and editing tools. However, there is an entirely different CMS genre, for which Zope also has been used. Other companies, such as struktur AG, used Zope successfully to interface with the XML Database Tamino (from software AG). The second common use is Zope as a Web-Application server, where it is used to build Web-based applications, such as online shops or project management tools. Of course, Zope is also suitable for regular Web sites.

And yet, there is a usage that we neglected so far. Zope can also be used as a reliable backend server managing the logistics of a company's operations. In fact, bluedynamics.com in Austria built a logistic software based on Zope 2 ZClasses and a relational database that was able to handle hundreds of thousands transactions each day from taking credit card information and billing the customer up to ordering the products from the warehouse using XML-RPC. In my opinion this is the true strength of Zope, since it allows not only Web-familiar protocols to talk to, but also any other network protocol you can imagine. Zope 3, with its component architecture, accelerates even more in this area, since third party products can be easily plugged in or even replace some of the defaults. For example the Twisted framework can replace all of ZServer (the Zope Server components).

Now that we have seen some of the common and uncommon uses of Zope it might be possible to formulate a more formal definition of Zope, just in case you are being asked at one point. *Zope is an application and backend server framework that allows developers to quickly implement protocols, build applications (usually Web-based) and function as glue among other net-enabled services.*

Before Zope was developed, Zope Corporation was reviewing many possible programming languages to develop the framework, such as Java, C/C++, Perl and Python. After extensive research they found that only Python would give them the competitive advantage in comparison to the other large framework providers, such as IBM, BEA and others.

Powerful Python

Python is a high-level object-oriented scripting language producing – by design – clean code through mandatory indentation. While Perl is also an interpreted scripting language, it lacks the cleanness and object-orientation of Python. Java, on the other hand, provides a nice object-oriented approach, but fails to provide powerful tools to build applications in a quick manner. So it is not surprising that Python is used in a wide variety of real world situations, like NASA, which uses Python to interpret their simulation data and connect various small C/C++ programs. Also, Mailman, the well-known mailing list manager, is being developed using Python. On the other hand, you have academics that use this easy-to-learn language for their introductory programming courses.

Since Python is such an integral part of the understanding of Zope, you should know it well. If you are looking for some introductory documentation, you should start with the tutorial that is available directly from the Python homepage `http://www.python.org/doc/current/tut/tut.html`. Also, there are a wide variety of books published by almost every publisher.

In the beginning there was...

Every time I am being asked to give a presentation or write a survey-like article about Zope, I feel the need to tell a little bit about its history, not only because it is a classic Open-Source story, but also because it gives some background on why the software behaves the way it does. So it should definitely not be missing here.

Before Zope was born, Zope Corporation (which was originally named Digital Creations) developed and distributed originally three separate products called Bobo, Principia and Aqueduct. Bobo was an object publisher written in Python, which allowed one to publish objects as pages on the web. It also served as object database and object request broker (ORB), converting URLs into object paths. Most of this base was implemented by Jim Fulton in 1996 after giving a frustrating Python CGI tutorial at the International Python Conference. Even though Bobo was licensed under some sort of "free" license, it was not totally Open-Source and Principia was the commercial big brother.

In 1998, Hadar Pedhazur, a well-known venture capitalist, convinced Digital Creations to open up their commercial products and publish them as Open Source under the name Zope. Zope stands for the "Z Object Publishing Environment". The first Zope release was 1.7 in December 1998. Paul Everitt, former CEO, and all the other people at Zope Corporation converted from a product company into a successful consultant firm. Alone the story how Zope Corporation went from a proprietary, product-based to a service-based company is very interesting, but is up to someone else to tell.

In the summer of 1999, Zope Corporation published version 2.0, which will be the base for the stable release until Zope 3.0 will outdate it in the next few years. Zope gained a lot of popularity with the 2.x series; it is now included in all major Linux distributions and many books have been written about it. Originally I was going to write this book on the Zope 2.x API, but with the beginning of the Zope 3.x development in late 2001, it seemed much more useful to do the documentation right this time and write an API book parallel to the development itself. In fact when these lines were originally written, there was no Zope Management Interface , and the initial security had just been recently implemented.

Zope 3 Components

Zope 3 will make use of many of the latest and hottest development patterns and technologies, and that with "a twist" as Jim Fulton likes to describe it. But Zope 3 also reuses some of the parts that were developed for previous versions. Users will be glad to find that Acquisition (but in a very different form) is available again as well as Zope Page Templates and the Document Template Markup Language - DTML (even though with less emphasis). Also, there is the consensus of a Zope Management Interface in Zope 3 again, but is completely developed from scratch in a modular fashion so that components cannot be only reused, but the entire GUI can be altered as desired.

But not only DTML, ZPT and Aquidition received a new face in Zope 3; external data handling has been also totally reworked to make external data play better together with the internal persistence framework, so that the system can take advantage of transactions, and event channels. Furthermore, the various external data sources are now handled much more generically and are therefore more transparent to the developer. But which external data sources are supported? By default Zope 3 comes with a database adaptor for Gadfly , but additional adapters for PostGreSQL and other databases already exist and many others will follow. Data sources that support XML-RPC, like the very scalable XML database Tamino, could also be seamlessly inserted. However, any other imaginable data source can be connected to Zope by developing a couple of Python modules, as described in various chapters.

During the last five years (the age of Zope 2) not only Zope was developed and improved, but also many third party products were written by members of the very active Zope community for their everyday need. These products range from Hot Fixes, Database Adaptors and Zope objects to a wide range of end user software, such as e-commerce, content management and e-learning systems. However, some of these products turned out to be generically very useful to a wide variety of people; actually, they are so useful, that they were incorporated into the Zope 3 core. The prime examples are the two internationalization and localization tools Localizer (by Juan David Ibáñez Palomar) and ZBabel (by me), whose existence shaped the implementation of the internationalization and localization support Zope 3 significantly. Another great product that made it into the Zope 3 core was originally written by Martijn Faassen and is called Formulator. Formulator allows the developer to define fields (representing some meta-data of a piece of content) that represent data on the one side and HTML fields on the other. One can then combine fields to a form and have it displayed on the Web. The second great feature Formulator came with was the Validator, which validated user-entered data on the server side. Formulator's concepts were modularized into schemas and forms/widgets and incorporated in Zope 3.

Altogether, the framework is much cleaner now (and more pythonic) and features that failed to make it into the Zope 2 core were incorporated.

Goals of this book

The main target audience for this book are developers that would like to develop on the Zope 3 framework itself; these are referred to as Zope developers in this book. But also Python programmers will find many of the chapters interesting, since they introduce concepts that could be used in other Python applications as well. Python programmers could also use this book as an introduction to Zope.

In general the chapters have been arranged in a way so that the Zope 3 structure itself could be easily understood. The book starts out by getting you setup, so that you can evaluate and develop

with Zope 3. The second part of the book consists of chapters that are meant as introductions to various important concepts of Zope 3. If you are a hands-on developer like me, you might want to skip this part until you have done some development. The third and fourth part are the heart of the book, since a new content component with many features is developed over a course of 12 chapters. Once you understand how to develop content components, part five has a set of chapters that introduce other components that might be important for your projects. The fifth part is intended for people that wish to use Zope technologies outside of Zope 3. The emphasis on testing is one of the most important philosophical transitions the Zope 3 development team has undergone. Thus the last chapter is dedicated to various ways to write tests.

Last but not least this book should encourage you to start helping us to develop Zope 3. This could be in the form of enhancing the Zope 3 core itself or by developing third party products, reaching from new content objects to entire applications, such as an e-commerce system. This book covers all the modules and packages required for you to start developing.

PART I

Zope 3 from a User's Point of View

This part concentrates on getting Zope 3 setup on your computer.

Chapter 1: Installing Zope 3

Before you can go on trying all the code snippets provided by the following chapters and recipes, you need to install Zope 3 on your system. This introduction chapter will only cover the basic installation and leave advanced setups for the following chapters to cover.

Chapter 2: The new Web-based User Interface

While this has not much to do with Python development, one must be comfortable with the GUI interface, so that s/he can test the features of the new code effectively. Furthermore, the TTW development interface will be briefly introduced.

Chapter 3: Install Zope Packages

A simple chapter for installing packages in your Zope 3 installation.

Chapter 4: Setting Up Virtual Hosting

This chapter describes how to setup virtual hosting in Zope 3, such as it is required for using Apache in front of Zope 3.

CHAPTER 1

INSTALLING ZOPE 3

Difficulty

Newcomer

Skills

- You should know how to use the command line of your operating system. (For Windows releases an Installer is provided.)
- You need to know how to install the latest Python successfully on your system.

Problem/Task

Before we can develop anything for Zope 3, we should install it of course.

Solution

1.1 Requirements

Zope 3 requires usually the latest stable Python version. For the Zope X3 3.0.0 release, this was Python 2.3.4 or better. Note that you should always use the latest bug-fix release. Zope 3 does not require you to install or activate any special packages; the stock Python is fine. This has the great advantage that you can use pre-packaged Python distributions (for example: RPM, deb, Windows Installer) of your favorite OS.

The only catch is that Zope 3's C modules must be compiled with the same C compiler as Python. For example, if you install the standard Python distribution

on Windows – compiled with Visual C++ 7 – you cannot compile Zope 3's modules with Cygwin. However, the problem is not as bad as it seems. The Zope 3 binary distributions are always compiled with the same compiler as the standard Python distribution for this operating system. On the other hand, if you want to compile everything yourself, you surely use only one compiler anyways.

On Un*x/Linux your best bet is `gcc`. All Zope 3 developers are using `gcc`, so it will be always supported. Furthermore all Linux Python distribution packages are compiled using `gcc`. On Windows, the standard Python distribution is compiled using Visual C++ 7 as mentioned above. Therefore the Zope 3 binary Windows release is also compiled with that compiler. However, people have also successfully used `gcc` using Cygwin – which comes with Python. Finally, you can run Zope 3 on MacOS X as well. All you need are the developers tools that provide you with `gcc` and the `make` program; everything you need. Both, Python and Zope 3, will compile just fine.

Python is available at the Python Web site[1].

1.2 Installing from SVN

In order to check out Zope 3 from SVN, you need to have a SVN client installed on your system, of course. If you do not have a SVN account, you can use the anonymous user to get a sandbox checked out:

```
svn co svn://svn.zope.org/repos/main/Zope3/trunk Zope3
```

After the checkout is complete, enter the Zope 3 directory:

```
cd Zope3
```

From there run `make` (so you need to have `make` installed, which should be available for all mentioned environments). If your Python executable is not called `python2.3` and/or your Python binary is not in the path, edit the first line of the `Makefile` to contain the correct path to the Python binary. Now just run make, which will build/compile Zope 3:

```
make
```

Copy `sample_principals.zcml` to `principals.zcml` and add a user with manager rights as follows:

```
1 <principal
2     id="zope.userid" title="User Name Title"
3     login="username" password="passwd" />
4
5 <grant role="zope.Manager" principal="zope.userid" />
```

[1] http://www.python.org/

▷ Line 2: Notice that you do not need "zope." as part of your principal id, but the id *must* contain at least one dot ("."), since this signals a valid id.

▷ Line 3: The login and password strings can be any random value, but must be correctly encoded for XML.

▷ Line 5: If you do not use the default security policy, you might not be able to use this `zope:grant` directive, since it might not support roles. However, if you use the plain Zope 3 checkout then roles are available by default.

Furthermore, during development you often do not want to worry about security. In this case you can simply give `anybody` the `Manager` role:

```
1 <grant role="zope.Manager" principal="zope.anybody" />
```

The fundamental application server configuration can be found in `zope.conf`. If `zope.conf` is not available, `zope.conf.in` is used instead. In this file you can define the types and ports of the servers you would like to activate, setup the ZODB storage type and specify logging options. The configuration file is very well documented and it should be easy to make the desired changes.

Now we are ready to start Zope 3 for the first time:

```
./bin/runzope
```

The following output text should appear:

```
------
2003-06-02T20:09:13 INFO PublisherHTTPServer zope.server.http (HTTP) started.
        Hostname: localhost
        Port: 8080
------
2003-06-02T20:09:13 INFO PublisherFTPServer zope.server.ftp started.
        Hostname: localhost
        Port: 8021
------
2003-06-02T20:09:13 INFO root Startup time: 5.447 sec real, 5.190 sec CPU
```

Once Zope comes up, you can now test the servers by typing the following URL in your browser: `http://localhost:8080/`. Test FTP using `ftp://username@localhost:8021/`. And even WebDAV is available using `webdav://localhost:8080/` in Konqueror.

An XML-RPC server is also built-in by default, but most objects do not support any XML-RPC methods, so that you cannot test it right away. See chapter "Availability via XML-RPC" for detailed instructions on how to use the XML-RPC server.

1.3 Installing the Source Distribution

1.3.1 Unpacking the Package

The latest release of Zope 3 can be found at `http://www.zope.org/Products/ZopeX3`. First download the latest Zope 3 release by clicking on the file that is available for all platforms, i.e. `ZopeX3-VERSION.tgz`. Use `tar` or WinZip to extract the archive; for example:

```
tar xzf ZopeX3-3.0.0.tgz
```

1.3.2 Building Zope

For the releases we provided the well-known "configure" / "make" procedure. So you can start the configuration process by using

```
./configure
```

If you wish to place the binaries of the distribution somewhere other than `/usr/local/ZopeX3-VERSION`, then you can specify the `--prefix` option as usual. Also, if you have Python installed at a non-standard location, you can specify the Python executable using `--with-python`. A full configuration statement could be

```
./configure --prefix=/opt/Zope3 --with-python=/opt/puython2.3/bin/python2.3
```

The immediately returned output is

```
Using Python interpreter at /opt/puython2.3/bin/python2.3

Configuring Zope X3 installation
```

Now that the source has been configured, we can build it using `make`. Type in that command. Only one line stating

```
python2.3 install.py -q build
```

will be returned and the hard drive will be busy for several minutes compiling the source. Once the command line returns, you can run the tests using

```
make check
```

Here both, the unit and functional tests are executed. For each executed test you have one "dot" on the screen. The check will take between 5-10 minutes depending on the speed and free cycles on your computer. The final output should look as follows:

```
Python2.3 install.py -q build
Python2.3 test.py -v
Running UNIT tests at level 1
Running UNIT tests from /path/to/ZopeX3-VERSION/build/lib.linux-i686-2.3
[some 4000+ dots]
----------------------------------------------------------------------
```

```
Ran 3896 tests in 696.647s

OK
```

The exact amount of tests run will depend on the version of Zope, the operating system and the host platform. If the last line displays an "OK", you know that all tests passed. Once you verified the check, you can install the distribution using

```
make install
```

1.3.3 Creating a Zope Instance

Once the installation is complete, Zope is available in the directory you specified in `--prefix` or under `/usr/local/ZopeX3-VERSION`. However, Zope will not yet run, since you have not created an instance yet. Instances are used for scenarios where one wants to host several Zope-based sites using the same base software configuration.

Creating a new instance is easy. Enter the Zope 3 directory and enter the following command:

```
/bin/mkzopeinstance -u username:password -d path/to/instance
```

This will create a Zope 3 instance in `path/to/instance`. A user having the login "username" and password "password" will be created for you and the "zope.manager" role is assigned to it. All the configuration for the created instance are available in the `path/to/instance/etc` directory. Please review all the information in there to ensure it fits your needs.

1.3.4 Run Zope

Zope is simply executed by calling

```
./bin/runzope
```

from the instance directory. The startup output will be equal to that of the source Zope SVN installation.

You are all done now! Once the server is up and running, you can test it via you favorite browser as described before.

1.4 Installing the Source Distribution on Windows without `make`

Installing the source distribution on Windows is possible even without `make`. However, you will need a supported C compiler to build the package. If you do not have

a C compiler or Cygwin installed, please use the Windows installer to install Zope 3. See in the next section for more details.

Before installing Zope 3, you should have installed Python 2.3.4 or higher. On Windows NT/2000/XP the extension `.py` is automatically associated with the Python executable, so that you do not need to specify the Python executable when running a script.

Once you unpacked the distribution, enter the directory. The software is built using

```
install.py -q build
```

Once the built process is complete, you can run the tests with

```
test.py -v
```

which should give you the same output as under Un*x/Linux. Once the tests are verified, the distribution is installed with the following command:

```
install.py -q install
```

You have now completed the installation of Zope 3. Follow now the final steps of the previous section to create an instance and starting up Zope.

Notice: This way of installing Zope 3 makes it really hard to uninstall it later, since you have to manually delete files/directories from various locations including your Python's `Lib\site-packages\` and `Scripts\` as well as completely removing the `zopeskel\` directory. If you use the Windows installer instead, an uninstallation program is provided and registered in the Control Panel's "Add/Remove Programs" applet.

1.5 Installing the Binary Distribution

Currently we only provide binary releases for Windows. These releases assume that you have the standard Windows Python release installed. The Windows binary release is an executable that will automatically execute the installer. Simply follow the instructions on the screen and Zope 3 will be completely installed for you.

You can later use the Control Panel's "Add/Remove Programs" applet to uninstall Zope 3 again.

CHAPTER 2

THE NEW WEB-BASED USER INTERFACE

Difficulty

Newcomer

Skills

- Some high-level object-oriented skills are needed for the examples to make sense.
- Familiarity with the component architecture would be useful, since some of the vocabulary would make more sense. Optional.

Problem/Task

At this point you might say: "I have installed Zope 3, but now what?" This is a good question, especially if you have never seen any version of Zope before. After Zope started with the bootstrap configuration, it starts up an HTTP and a FTP server. Via the HTTP server, a Web user interface is provided in which the site manager cannot only configure the server further, but also develop so called "through-the-Web software" (short: TTW software). After introducing the basic elements and concepts of the Web interface, which is known as the ZMI (Zope Management Interface), a couple simple demonstrations are given. The Zope X3 3.0.0 release concentrated mainly on filesystem-based development – which this book is about – so that TTW is still very immature and not even available via the distribution.

Solution

After Zope 3 started, you can enter the Zope Management Interface (ZMI) via the "manage" screen. The full URL is then `http://localhost:8080/manage`.

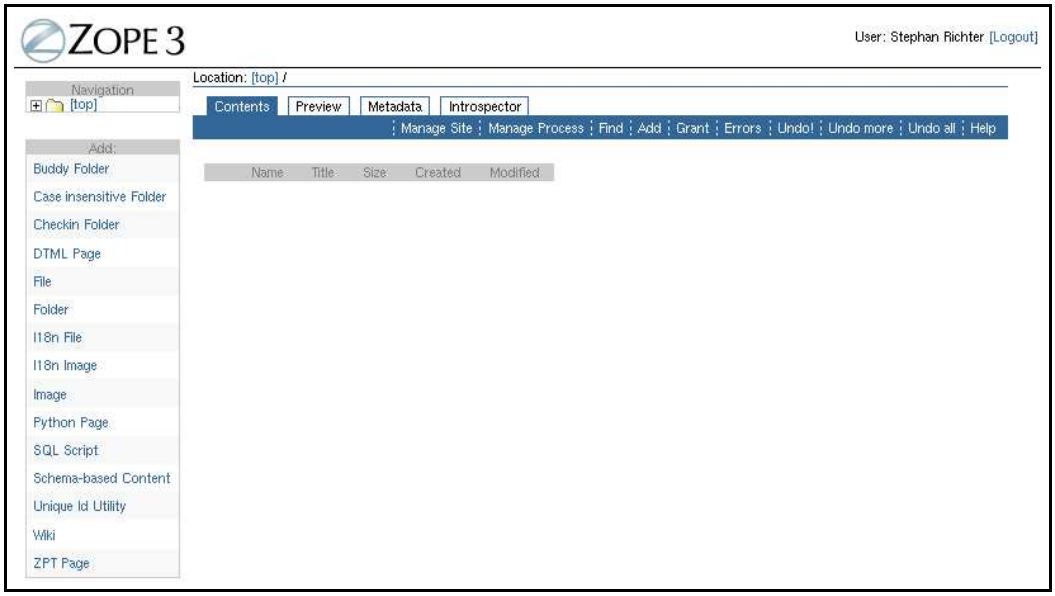


Figure 2.1: The Initial Contents View of the Root Folder

Zope 3 has a very flexible skinning support, which allows you to alter the look and a little bit the feel of the ZMI. Other skins can be reached using the `++skin++` URL namespace. One of the other nice skins is "ZopeTop" – it is excluded from the Zope X3 3.0.0 release but available in the repository.

This book will exclusively use the default skin to describe the UI.

2.1 Getting comfortable with the ZMI

The ZMI is basically a classic two column layout with one horizontal bar on the top. The bar on the top is used for branding and some basic user information. The thin left column is known as the "Navigators" column. The first box of this column is usually the navigation tree, which is just a hierarchical view of the object database structure. This works well for us, since the two main object categories are normal objects and containers. Below the tree there can be a wide range of other boxes, including "Add:".

The main part of the screen is the wide right column known as the "workspace". The workspace specifically provides functionality that is centric to the accessed ob-

Figure 2.2: The Initial Contents View using the `ZopeTop` skin.

ject. On the top of the workspace is the full path of the object in "breadcrumbs" form, which provides a link to each element in the path.

Below the location, a tab-box starts. The tabs, known as "ZMI views", are various different views that have been defined for the object. You can think of these views as different ways of looking at the object. A good example is the "Contents" view of a `Folder` object.

Below the tabs, you see another list of actions, known as "ZMI actions". ZMI actions are also object specific, but usually they are available in a lot of different objects. Common actions include "Undo" , "Find" , "Grant" and "Help" which are available on all objects.

Below the actions is the "viewspace", which may contain several elements. All views have the "content", which contains the information and forms of the selected tab. On the right side of the viewspace there can be an optional column called "context information". It is sometimes used to display view-specific help or metadata.

Overall, the entire ZMI is built to contain these elements for consistency.

So, let's do something with the ZMI. Our goal will be to create a Folder and write a ZPT Page that displays title of the folder. New objects can be added to the root by clicking on "Add" of the ZMI actions. You will be now presented with a list of all available content objects. Select the "Folder" and insert the name "folder" in the input box on the bottom. Finalize the addition by clicking on the "Add" button. The system will return you to the root folder's contents view. To add a sensible

Figure 2.3: Marked up ZMI Screenshot Pointing Out all the Sections

title to our new folder, click on the empty cell of the "Title" column. An input field should appear in which you may add the title "Some Cool Title" and hit enter.

Now enter the folder, and add a "ZPT Page" called "showTitle.html" the same way you added the folder. Now go and edit the page and add the following code:

```
1 <html>
2   <body>
3     <h4>Folder Title</h4>
4     <br/>
5     <h1 tal:content="context/zope:title">title here</h1>
6   </body>
7 </html>
```

The strange `tal` namespace on Line 5 is the first appearance of Zope's powerful scripting language, Page Templates. The `content` attribute replaces "title here" with the title of the folder which is accessed using `context/zope:title`.

If you now click on the page's "Preview" tab, you will see the title of the folder being displayed. Of course you can just open the page directly using `http://localhost:8080/folder/showTitle.html`.

2.2 Help and Documentation

Zope 3 comes with an extensive online help system. Generally, it can be reached either via the explicit "Help" link in the ZMI actions or via the context-sensitive help box, which pops up on the right side of the viewspace, if help screens are available.

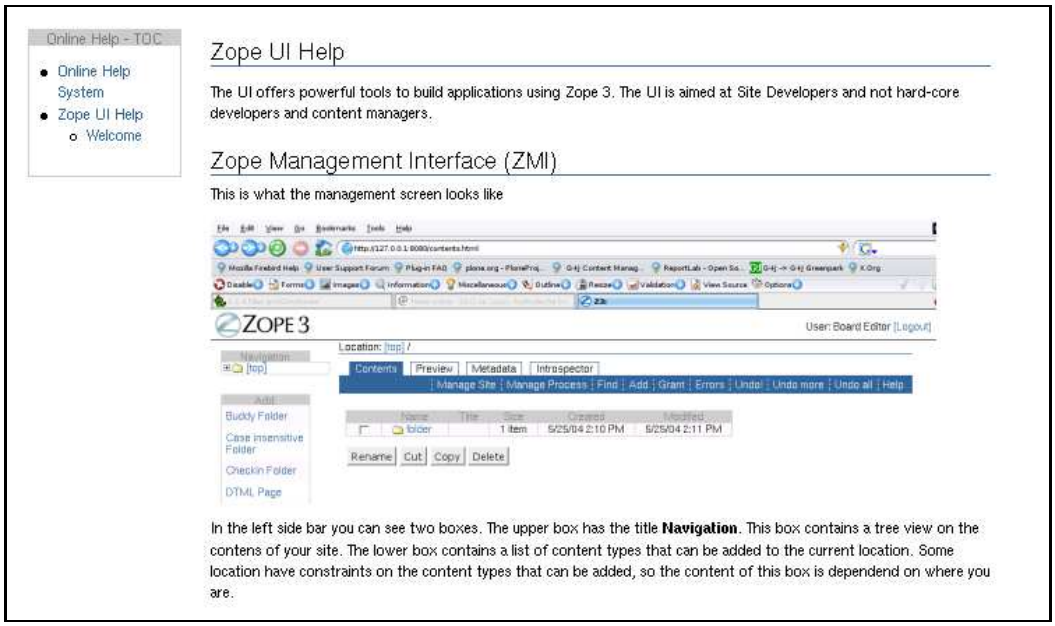


Figure 2.4: The Online Help showing a Table of Contents on the left.

Another helpful feature is the interface introspector. You might have already noticed the "Introspector" tab that is available for all objects. It provides a list of all interfaces that the object provides and base classes that the object extends.

The interfaces and classes listed in the introspector are actually linked to the API documentation, which is third major dynamic documentation tool in Zope 3. The API Doc Tool is commonly accessed via `http://localhost:8080/++apidoc++` and provides dynamically-generated documentation for interfaces, classes, the component architecture and ZCML.

Clearly, these tools are not just useful for the scripter or user of the Web interface, but also for Python developers to get a fast overview of how components are connected to each other and the functionality they provide.

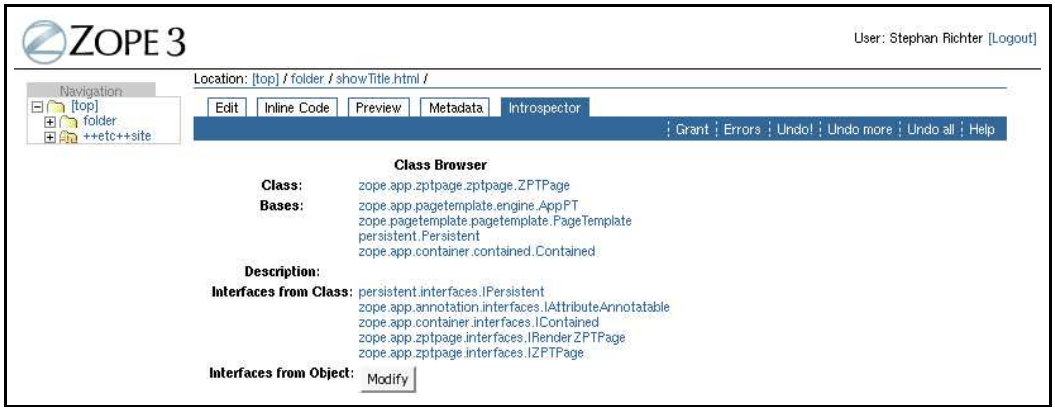


Figure 2.5: The Introspector view for an object.

2.3 The Scripter's World

While Zope 3 is truly an object- and component-oriented application server, we are trying to accomedate the scripter – someone who is a high-level developer that is not very familiar with advanced programming concepts – as well. Several tools have been developed for the scripter and are available in the Zope 3 source repository. However, for the Zope X3 3.0.0 release we decided to concentrate on providing the Python developer with a solid framework to work with. The focus of the upcoming releases will be provide facilities that help scripters to switch fromPHP, ColdFusion and ASP to Zope 3. Furthermore, we will provide a direction to migrate a scripter to become a developer.

2.4 Software versus Content Space

Zope 2 did a great mistake by allowing software and content to live in the same place. We did not want to repeat the same mistake in Zope 3. Therefore we developed a "content space" and "software space" . So far we have only worked in content space.

2.4.1 Content Space

As the name implies, in content space we only store content. While we think that simple DTML and ZPT scripts are dynamic content, Python scripts are clearly software and are therefore not available in content space. The only type of programming that is supported here is what we consider scripter support. However, being a

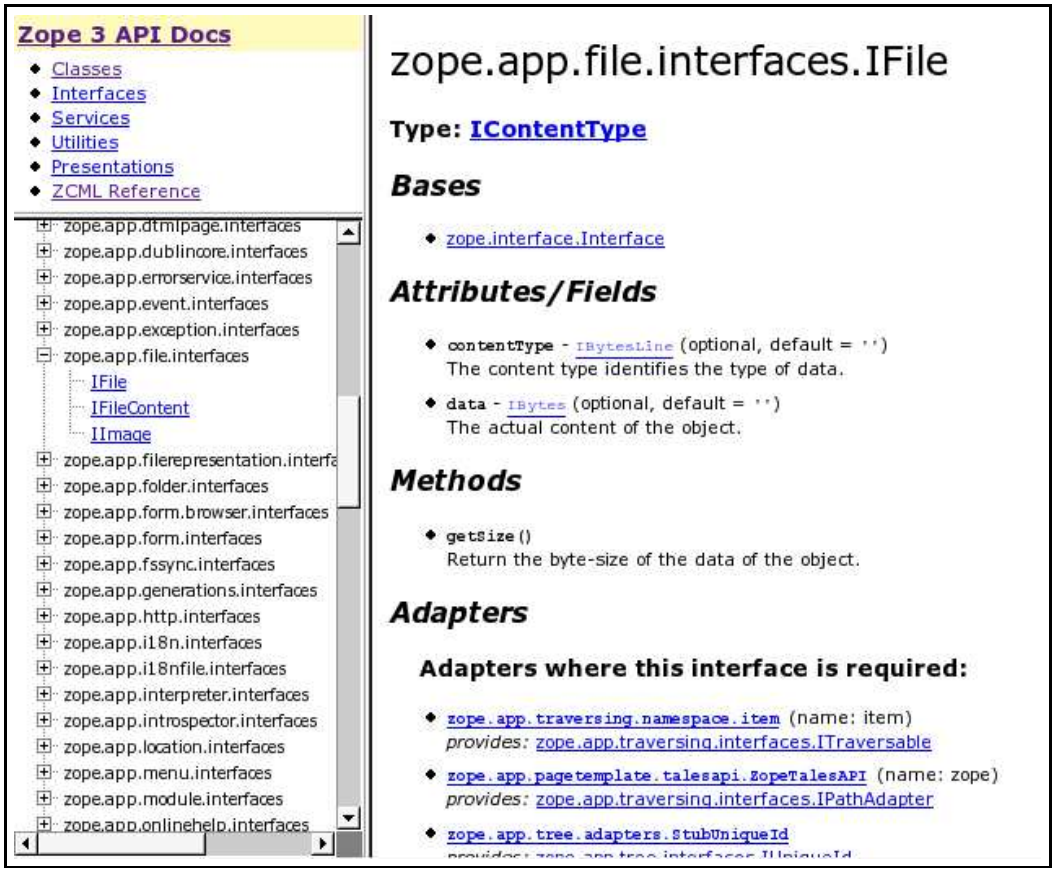


Figure 2.6: The API Doc tool showing the Interfaces menu and the IFile details.

"scripter" in this sense is not a desirable state and we want to provide a way to help the scripter become a TTW developer .

2.4.2 Software Space

Every "Site" object has a software space, which can be accessed via the "Manage Site" action. You can see the link when looking at the contents of the root folder, since the root folder is always a site. But how do you create one; it is not on the list of addable types? To create a site, create a folder and click on the "Make a site" action item. That's it! Any folder can upgrade to become a site.

Once you have clicked on "Manage Site", you are presented with an overview of possible tasks to do. Each site consists of several packages, including the `default` package, which is always defined. Packages are used to later pack up and distribute

software. An involved registration and constraint system ensures the integrity of the setup.

If you now click on the "Visit default folder" action link, you end up in the contents view of the package. The goal is that you add local components here that act as software. Since this looks and behaves very similar to the folder's content space's "Contents" view, we often call the added components "meta-content".

Okay, let's see how all this hangs together. The simplest example is to create a local Translation Domain and use it to translate a silly message. Assuming that you are already in the contents view of the default package, click on "Add", select "Translation Domain"and enter "Translations" as the name; press the "Add" button to submit the request. You will now be prompted to register the component. Create a registration by clicking on "Register". Register the domain under the name "silly" (which will be the official domain name). The provided interface should be `ILocalTranslationDomain` and the registration status is set to "Active". Finish the request by clicking on the "Add" button.

Now you have an active Translation Domain utility. Click on the "Translate" tab to enter the translation environment. The first step is to add some languages. Enter `en` under "New Language" and press that "Add" button. Do the same for `de`. Now you have to select both, English and German from the list on the left and click "Edit". You will see that the table further down on the page grew by two more columns each representing a language. To add a new translation, look at the first row. For the Message Id, enter "greeting", for the `de` and `en` column "Hallo Welt!" and "Hello World!", respectively. To save the new translation hit the "Edit Messages" button at the bottom of the screen. You now have a translation of the greeting into English and German.

Now that we have defined our software – the Translation Domain – it is time to go back to content space and "use" the new software. In the root folder create a new ZPT Page called "i18n.html" and add the following content:

```
1 <html>
2   <body i18n:domain="silly">
3     <h1 i18n:translate="">greeting</h1>
4   </body>
5 </html>
```

Once you save these lines and hit the "Preview" tab, you will see a big "Hello World!" on your screen, assuming that you have English as the preferred language. If you change the language now to German (de), then "Hallo Welt!" should appear.

In Mozilla you can change the preferred language under "Edit", "Preferences...", "Navigator" and "Languages". Simply add "German [de]" and move it to the top of the list.

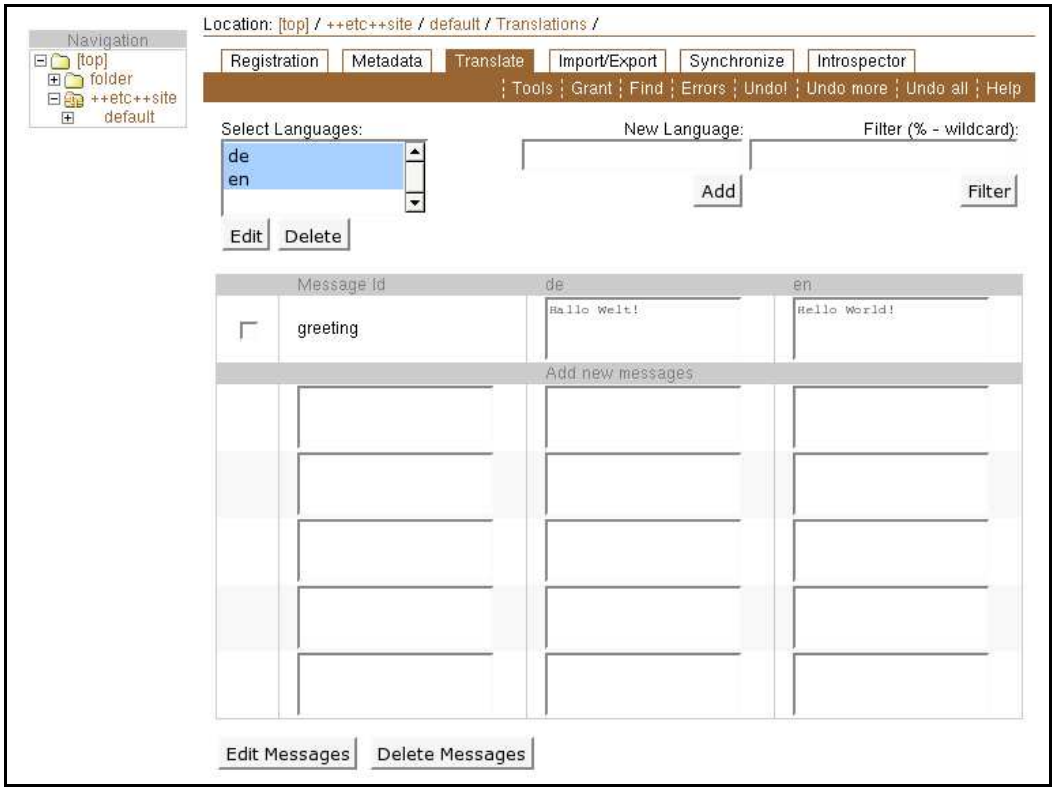


Figure 2.7: The "Translate" screen of the Translation Domain utility.

We just saw how simple it is to develop software in the Site and then use it in Content space. As Zope 3 matures you will much more impressive features, such as TTW development of content component using Schemas and writing new views.

This concludes our little diversion into the lands of through-the-web development. After talking a little bit more about setting up Zope, we will turn to filesystem-based development and start giving a more formal introduction to the concepts in Zope 3.

CHAPTER 3

INSTALL ZOPE PACKAGES

Difficulty

Newcomer

Skills

- You should know how to use the command line of your operating system.
- You should know how to install Python and Zope 3 (and do it before reading this chapter).

Problem/Task

After having installed Zope 3, there is only so much you can do with it. However, there is a bunch of interesting 3rd party add-on packages, such as relational database adapters and a Wiki implementation. This chapter will demonstrate how to install these packages, especially the ones maintained in the Zope repository.

Solution

Installing 3rd party packages in Zope 3 is much more pythonic and explicit than in Zope 2. A Zope 3 add-on package can be put anywhere in your Python path like any other Python package. This means that you can use the `distutils` package to distribute your add-on.

However, once you installed the package, the Zope 3 framework does not know about it and no magical package detection is used to find available packages. The Zope 3 framework is configured using the Zope Configuration Markup Language (ZCML). Zope-specific Python packages also use ZCML to register their components

with Zope 3. Therefore we will have to register the packages root ZCML file with the startup mechanism.

In this chapter I will demonstrate how to install the "Wiki" for Zope 3 application. If you are using the repository version of Zope 3, it will be already installed, but the steps are the same for all other packages as well.

3.1 Step I: Determining the Installation Directory

Determining the place to install a package can be one of the most difficult challenges, since packages can be anywhere in the path hierarchy. For example, Jim Fulton's Buddy Demo package is a top-level Python package, whereby the Wiki application lives in `zope.app` and the Jobboard Example in `zope.app.demo`.

However, we usually want to place any 3rd party packages in the common Zope 3 directory structure, since it will be easier to find the packages later. Once we have determined the Zope 3 package root we are all set. For the repository distribution the package root is `Zope3/src`, where `Zope3` is the directory that you checked out from the repository, i.e. `svn://svn.zope.org/repos/main/Zope3/trunk`. For distributions the package root is `Zope3/lib/python`, where `Zope3` is `/usr/local/ZopeX3-VERSION` by default or the directory specified using the `--prefix` option of `configure`.

Since we are installing the Wiki application, we have to go to the `Zope3/zope/app` directory, which should be available in every Zope installation.

3.2 Step II: Fetching the Wiki application from SVN

The next step is to get the package. Usually you probably just download a TAR or ZIP archive and unpack it in the directory. However, for the Wiki application there is no archive and we have to fetch the package from SVN.

Assuming you have the SVN client installed on your computer, you can use the following command to do an anonymous checkout of the `wiki` package from the Zope X3 3.0 branch:

```
svn co \
  svn://svn.zope.org/repos/main/Zope3/branches/ZopeX3-3.0/src/zope/app/wiki \
  wiki
```

While SVN allows you to name the created directory whichever way you want to, it is necessary to name the directory `wiki`, since imports in the package assume this name. Once the command line returns, the package should be located at `Zope3/zope/app/wiki`.

3.3 Step III: Registering the Package

The next step is to register the package's new components with the Zope 3 framework. To do that we have to place a file in the `package-includes` directory. In a repository hierarchy this directory is found in `Zope3`. In the distribution installation, it is located in `Zope3/etc`. Enter this directory now and add a file called `wiki-configure.zcml` with the following content:

```
1 <include package="zope.app.wiki" />
```

The `package-includes` directory is special in the sense that it executes all files ending on `-meta.zcml` will be executed when all meta directives are initiated and all files with `-configure.zcml` are evaluated after all other configuration was completed.

Once you save the file, Zope will be able to evaluate the Wiki package's configuration.

3.4 Step IV: Confirm the Installation

If Zope 3 is running, stop it. Start Zope 3 and read the messages it prints during startup. If it starts without any error messages, your package has been successfully installed.

Note that Zope 3 has no way (contrary to Zope 2) to show you which products have been successfully installed, since there is no formal concept of a "Zope add-on". However, you usually can tell immediately about the success of an installation by entering the ZMI. If the "Wiki" object is available as a new content type, then you know the configuration was loaded correctly.

3.5 Step V: Add a Sample Wiki

In a web browser, visit your Zope 3 web server by entering the URL `http://localhost:8080/` or similar, depending on your browser and whether Zope 3 installed on the local machine. Click on the "top" folder in the Navigation box (upper left). To add a sample job board:

- Click "Add" in the actions menu.
- Select "Wiki" from the list of content types.
- Type "wiki" (without quotes) in the text field and click the "Add" button.
- Since the Wiki object provides a custom adding screen, you have to confirm the addition once more by clicking the "Add" button.

You should now see a Wiki entry in the list of objects. To experiment with the object, simply click on "wiki".

That's it. As you can see, the installation of a Zope 3 add-on package is equivalent to the Python way with the addition that you have to point Zope 3 to the ZCML file of the package.

CHAPTER 4

SETTING UP VIRTUAL HOSTING

Difficulty

Newcomer

Skills

- Have a basic understanding of the namespace syntax in Zope URLs.
- You should be familiar with the Apache configuration file.

Problem/Task

One of the most common tasks in the Zope world is to hide Zope behind the Apache Web Server in order to make use of all the nice features Apache provides, most importantly SSL encryption.

Solution

Apache and other Web servers are commonly connected to Zope via rewrite rules specified in virtual hosts. It is Zope's task to interpret these requests correctly and provide meaningful output. You might think that this is easy since you just have to point to the right URL of the Zope server. But this is only half the story. What about URLs that point to another object? At this point you need to tell Zope what the true virtual hosting address is. In Zope 3 this is accomplished using a special namespace called `vh`, which specifies the "public" address.

Before we can start setting up a virtual hosting environment on our server, we should complete the following checklist:

1. Make sure Zope 3 is running at `http://localhost:8080/site/` or more generically at `http://destination_url:port/path-to-site/`.
2. Make sure Apache is running at `http://www.example.com:80/` or more generically at `http://public_url:port/`

Zope 3 uses its URL namespace capability to allow virtual hosting, so that no special component or coding practice is required, which means virtual hosting is always available. Generally, namespaces are specified using `++namespace++` as one element of the URL. For the `vh` namespace we have `++vh++Public-URL++`. Note that the `++` at the end of the URL is specific to the `vh` namespace. It signalizes the end of the public URL.

The namespace approach has the advantage that one can never lock her/himself out due to misconfiguration. Some Zope 2 virtual hosting solutions had this problem and caused unnecessary headaches. In Zope 2 one also had to add an additional object. Zope 3 is not using any service or utility for this task, which makes virtual hosting support a very core functionality.

However, from an Apache point of view, the setup is very similar to Zope 2. In the `httpd.conf` file – usually found somewhere in `/etc` or `/etc/httpd` – insert the following lines:

```
1  LoadModule proxy_module /path/to/apache/1.3/libproxy.so
2
3  Listen 80
4
5  NameVirtualHost *:80
6
7  <VirtualHost *:80>
8    SSLDisable
9    ServerName www.example.com
10   RewriteEngine On
11   RewriteRule ^/site(/?.*) \
12 http://localhost:8080/site/++vh++http:www.example.com:80/site/++$1 \
13 [P,L]
14   CustomLog /var/log/apache/example.com/access.log combined
15   ErrorLog /var/log/apache/example.com/error.log
16 </VirtualHost>
```

▷ Line 1: Load the module that allows rewriting and redirecting the URL.

▷ Line 3: Setup the Apache server for the default port 80.

▷ Line 5: Declare all incoming request on port 80 as virtual hosting sites.

▷ Line 7-16: These are all specifc configuration for the virtual host at port 80.

▷ Line 8: Do not use SSL encryption for communication. We'll only allows normal HTTP connections.

- Line 9: The virtual host is know as `www.example.com` to the outside world.
- Line 10: Turn on the Rewrite Engine, basically telling Apache that this virtual host will rewrite and redirect requests.
- Line 11-13: The code on these lines should be really in one line. It defines the actual rewrite rule. The rule says:

  If you find the URL after the hostname and port to begin with `/site`, then redirect this URL to `http://localhost:8080/site/++vh++http:www.example.com:80/site/++` plus whatever was behind `/site`.

  Example: `www.example.com:80/site/hello.html` is rewritten to `http://localhost:8080/site/++vh++http:www.example.com:80/site/++/hello.html`.

  Note that the part after `++vh++` must strictly be of the form ⟨`protocol`⟩`:`⟨`host`⟩`:`⟨`port`⟩`/`⟨`path`⟩. Even if the port is 80 you have to specify it.
- Line 14: Defines the location of the access log.
- Line 15: Defines the location of the error log.

And we are done. It's easy, isn't it? All you need to do is to restart Apache, so that the changes in configuration will take effect.

There is nothing special that needs to be configured on the Zope 3 side. Zope is actually totally unaware of the virtual hosting setup. Note that you do not have to map the URL `www.example.com/site` to `localhost:8080/site` but choose any location on the Zope server you like.

You can now combine the above setup with all sorts of other Apache configurations as well, for example SSL. Just use port 443 instead of 80 and enable SSL.

Current Problems: The XML navigation tree in the management interface does not work with virtual hosting, because of the way it treats a URL.

Thanks to Marius Gedminas for providing the correct Apache setup.

PART II

The Ten-Thousand Foot View

In this part of the book an overview over some of the fundamental ideas in Zope 3 is given without jumping into the technical detail. The reader might prefer to read these recipes after reading some of the more hands on sections, like the ones that follow.

Chapter 5: The Zope 3 Development Process

This chapter briefly introduces the process that is used to develop and fix components for Zope 3.

Chapter 6: An Introduction to Interfaces

Since Interfaces play a special role in Zope 3, it is important to understand what they are used for and what they offer to the developer.

Chapter 7: The Component Architecture – An Introduction

An overview over components and their possible interaction.

Chapter 8: Zope Schemas and Widgets (Forms)

One of the powerful features coming up over and over again in Zope 3, are schemas, an extension to interfaces that allows attributes to be specified in more detail allowing auto-generation of forms as well as the conversion and validation of the inputted data.

Chapter 9: Introduction to ZCML

While you are probably familiar with ZCML by now, it is always good to review.

Chapter 10: I18n and L10n Introduction

This introduction presents the new I18n and L10n features of Zope 3 and demonstrates how a developer can completely internationalize his/her Zope application. I18n in ZPT, DTML, ZCML and Python code will be discussed.

Chapter 11: Meta Data and the Dublin Core

Everyone knows about it, everyone talks about it, but few know the details. Naturally, the fields of the DC will be discussed as well of how they can and should be used.

Chapter 12: Porting Applications

For many developers it is very important to have a migration path from their Zope 2 applications to Zope 3. This chapter will concentrate on the technical aspects and discuss the design changes that have to be applied to make an old Zope application fit the new model. The port of ZWiki will be used as an example.

CHAPTER 5

THE ZOPE 3 DEVELOPMENT PROCESS

Difficulty

Newcomer

Skills

- Be familiar with Python's general coding style.

Problem/Task

The simple question this chapter tries to answer is: "How can I participate in the Zope 3 development?" However, the Zope 3 developers strongly encourage 3rd party package developers to adopt the same high standard of code quality and stability that was used for Zope 3 itself.

Solution

Since Zope 3 was developed from scratch, there was much opportunity; not only in the sense of the software, but also in terms of processes, organization and everything else around a software project.

Very early on, it was decided that the Zope 3 project would provide a great opportunity to implement some of the methods suggested by the eXtreme Programming development paradigm. The concept of "sprints" was introduced in the Zope development community, which are designed to boost the development and introduce the new framework to community members. Other changes include the establishment of a style guide and a unique development process. It is the latter that I want to discuss in this chapter, since it is most important to the actual developer.

5.1 From an Idea to the Implementation

When you have a big software project, it is necessary to have some sort of (formal) process that controls the development. Such a process stops developers from hacking and checking in half-baked code, which requires a lot of discipline among free software developers, since there is often no compensation. With the start of the Zope 3 development, we tried to implement a flexible process that can be adjusted to the task at hand.

Once a developer has an idea about a desired feature, he usually presents the idea on the Zope 3 developers mailing list or on IRC. Here the developer can figure out whether his feature already exists or whether the task the feature seeks to complete can be accomplished otherwise. If the idea is a good one, then one of the core developers usually suggests to write a formal proposal that will be available in the Zope 3 Proposals wiki. Once the developer has written a proposal, s/he announces it to the mailing list for discussion. Comments to the proposals are usually given via E-mail or directly as comments on the wiki page. The discussion often requires changes to be made to the proposal by adjusting the design or thinking of further use-cases. Once a draft is approved by the other developers – silence is consent, though Jim Fulton normally likes to have the last say – it can be implemented. While we tried hard not to make the proposal writer the implementor, it works out that the proposal writer almost always has to implement the proposal.

But how can you get the developer "status" for Zope 3? Besides the community acceptance, there are a couple formal steps one must take to receive checkin rights. The first step is to sign the "Zope Contributor Agreement" , which can be found at `http://dev.zope.org/CVS/Contributor.pdf`. Once you have signed and sent this document to Zope Corporation, you can deposit your SSH key online for secure access. All this is described in very much detail on the `http://dev.zope.org/CVS/` wiki pages.

As developer you usually check in new features (components) or bug fixes. Both processes are slightly different, since a bug fix is much less formal.

5.1.1 Implementing new Components

Let's say you want to implement a new component and integrate it into Zope 3. For the development of new software eXtreme Programming provides a nice method, which we adopted for Zope 3 and is outlined in the following paragraphs.

Starting with your proposal, you develop the interface(s) for your component. Often you will have written the interfaces already in the proposal, since it helps explaining the functionality. If not, use the text of the proposal for documenting

your interfaces and its methods. For a detailed introduction to interfaces see the next chapter, which covers the formal aspects of writing interfaces.

Next, one should write the tests that check the implementation and the interfaces themselves. When testing against interfaces, one can use so-called stub-implementations of the interfaces. See chapter "Writing Tests against Interfaces". If you are not certain about implementation, use prototype code which is thrown away afterwards. Note: I found that this step usually requires the most understanding of Zope 3, so that most of the learning process happens at this point. However, it is also the point were most new developers get frustrated. See the "Writing Tests" part in general for an extended discussion about tests.

Now, you are ready to write the implementation of the interfaces. There is a nice little tool called `pyskel.py` (found in the `ZOPE3/utilities` folder) that will help you to get a skeleton for the class implementing a specified interface. You can get hands-on experience in the "Content Components – The Basics" part.

The final step is to run the tests. Start out by running the new tests only and, after everything passes, all of the Zope 3 tests as confirmation. Once all of Zope 3's tests pass, you can check-in the code. It is a good idea to ask the other developers to update their sandboxes and review your code and functionality.

5.1.2 Fixing a bug

In order to effectively fix bugs, you need to have supporter status for the Zope 3 Bug and Feature Collector, so that you can change the status of the various issues you are trying to solve. You can also ask an existing supporter of course, though this is much more cumbersome. Contact the Zope 3 mailing list (`zope3-dev@zope.org`) to get this status and they will help you.

Once you have access, accept an issue by assigning it to you in the Collector. At this point noone else will claim the issue anymore. The first step is to create tests that clearly point out the bug and fail due to the bug. Now try to fix the bug. While fixing you might discover other untested functionality and side-effects, so it is common to write more tests during the "fixing" process.

Finally, similar to the development of new components, you should run the new/local tests first, see whether they pass and then run the global tests. It sometimes happens that you will not be able to solve a bug, since tests of other packages will fail that you do not understand. At that stage, you should create a branch and ask other developers for help. Once you are done with the code and all tests pass, check in the changes and ask people to have a look. Once you are more experienced, a code review will not be necessary anymore.

5.2 Zope 3 Naming Rules

Zope 2 was a big mess when it comes to the naming of classes, methods and files. It was almost impossible to find a method in the API. I was always very impressed by the well-organized and consistent Java API, where you often just "guess" a method name without looking it up. Therefore the Zope 3 developers introduced a strict naming style guide that is maintained as part of the Zope 3 development wiki. In retrospect the guide brought us a lot of cleanness to the code base and made it much easier to remember Zope 3's APIs.

In the following sub-sections I will give you an overview of these conventions. See `http://dev.zope.org/Zope3/CodingStyle` for the detailed and up-to-date guide.

5.2.1 Directory Hierarchy Conventions

First of all, package and module names are all lowercase. They should be also kept short, so that two consecutive words should be rare. If they appear, just put them together; use no underscore.

The top-level directories are considered packages, like `zodb` or `zope.i18n` for example. There is a special package called `zope.app` that contains the Zope 3 application server. It is special because it contains application server specific sub-packages, which can be distributed separately. Each distribution package contains an `interfaces` module (depending on your needs it can be implemented as a file-based module or as package). This module should capture all externally available interfaces. Local interfaces that will be only implemented once can live in the same module with its implementation.

Usually, presentation-specific code lives in a separate module in the package; therefore you will often see a `browser` directory or `browser.py` file in the package. If the package only defines a few small views a file is preferred, as it is usually the case for XML-RPC views, since you usually have only a couple of classes for the XML-RPC support.

There are no rules on where a third party package must be placed as long as it is found in the Python path. Some developers prefer to place additional packages into `ZOPE3/src` and make them top-level packages. Others place their packages in `ZOPE3/src/zope/app` to signalize the package's dependency on the Zope application server.

5.2.2 Python Naming and Formatting Conventions

While the generic Python style guide allows you to *either* use spaces or tabs as indentation, we only use four spaces in Zope for indentation, since otherwise the code base might be corrupted, which can cause a lot of grief.

For method and variable naming we use exclusively Camel case naming (same as Java) throughout the public API. Private attribute names can contain underscores. For other usages of the underscore character (_), see PEP 8, the Python style guide. The underscore character is mainly used in method and variable names to signify private and protected attributes and methods (including built-in ones, like `__init__`). Classes always start with upper case with exception of ZCML directives, which is a special case. Attribute and method names always begin with a lower letter, whereby a method should always start with a verb. Here is an example of these rules:

```
1 class SimpleExample:
2
3     def __init__(self, text):
4         self.text = text
5
6     def getText(self):
7         return self.text
```

Additionally, for legal reasons and to protect the developers' rights, the Zope 3 community requires a file header in every Python file. Excluded are empty `__init__.py` files. Here is the standard ZPL file header:

```
1  ##############################################################################
2  #
3  # Copyright (c) 2004 Zope Corporation and Contributors.
4  # All Rights Reserved.
5  #
6  # This software is subject to the provisions of the Zope Public License,
7  # Version 2.1 (ZPL).  A copy of the ZPL should accompany this distribution.
8  # THIS SOFTWARE IS PROVIDED "AS IS" AND ANY AND ALL EXPRESS OR IMPLIED
9  # WARRANTIES ARE DISCLAIMED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
10 # WARRANTIES OF TITLE, MERCHANTABILITY, AGAINST INFRINGEMENT, AND FITNESS
11 # FOR A PARTICULAR PURPOSE.
12 #
13 ##############################################################################
14 """A one line description of the content of the module.
15
16 If necessary, one can write a longer description here. This is also a
17 good place to include usage notes and requirements.
18
19 $Id$
20 """
```

▷ Line 3: The copyright notice should always contain all years this file has been worked on. At the beginning of each year we use a script to add the new year to the file header.

- Line 1–13: Only use this header, if you are the only author of the code or you have the permission of the original author to use this code and publish it under the ZPL 2.1. Just to be on the safe side, always ask Jim Fulton about checking in code that was not developed by yourself. Of course, you must be yourself willing to publish your code under the ZPL. Note that you do *not* need to sign a contributor agreement to add this header, unless you want to add the code in the zope.org source repository. Also, the ZPL 2.1 does not automatically make Zope Corporation a copyright owner of the code as well, as it was the code for ZPL 2.0.
- Line 19: This is a place holder for the source code repository to insert file revision information, which can be extremely useful.
- Line 14–20: The module documentation string should always be in a file. The first line of the doc string is a short description of the module. Next, an empty line is inserted after which you can give more detailed documentation of the module. For example, in executable files, you usually store all of the help in this doc string.

In general, you should document your code well, so that others can quickly understand what it does. Feel free to refer to the interface documentation.

Interfaces have a couple more naming constraints. The name of an interface should always start with a capital "I", so that they can be easily distinguished from classes. Also, interface declarations are used for documenting the behavior and everything around the component, so that you should have almost all your documentation here. Zope 3's API documentation tool mainly uses the interface doc strings for providing information to the reader.

5.2.3 ZCML Naming and Formatting Conventions

ZCML is a dialect of XML, so that the usual good practices of XML apply. ZCML sub-elements are indented by two spaces; again, let's not use tab characters, as their width is ambiguous. All attributes of an element should be each on a separate line with an indentation of four spaces.

```
1  <configure
2      namespace="http://namespaces.org/my-namespace">
3
4    <namespace:directive
5        attr="value">
6
7      <namespace:sub-directive
8          attr1="value"
9          attr2="This is a long value that
10                spans over two lines."
11         />
12
13   </namespace:directive>
```

```
14
15 </configure>
```

While it is possible to specify any amount of ZCML namespaces inside a `configure` element, you should only declare the namespaces you actually use, like you only import objects in Python that the code below works with. As for all source code in Zope, we try to avoid long lines greater than 80 characters. However, sometimes you will notice that this is not possible, since the dotted Python identifiers and ids can become very long. In these rare cases it is okay to have lines longer than 80 characters.

5.2.4 Page Template Naming and Formatting Conventions

Page Templates were designed, like the name says, to provide presentation templates. Unfortunately, it is possible to supply Python-based logic via TALES expressions as well. In Zope 3 templates we avoid Python code completely, since the necessary logic can be provided by an HTML Presentation component as seen in the "Adding Views" chapter. In general, even if you have complex non-Python logic in a template, try to think about a way to move your logic into the Python-based presentation class.

Finally, HTML is a subset of XML, which means that all contents of double tags should be indented by two spaces (as for ZCML). For tag attributes we use four spaces as indentation. Remember that it is very hard to read and debug incorrectly indented HTML code!

5.2.5 Test Writing Conventions

While not necessary for the standard Python `unittest` package, the Zope 3 test runner expects unit tests to be in a module called `tests`. If `tests` is a directory (package), then you should not forget to have an empty `__init__.py` file in the directory. Similarly, you should have an `ftests` module for functional tests .

Usually you have several testing modules in a `tests` directory. All of these modules should begin with the prefix `test_`. Note that this is one of the few violations to not use the underscore character in a module name. The prefix is very helpful in detecting the actual tests, since sometimes supporting files are also located in the `tests` directory.

The testing methods inside a `TestCase` do not have doc strings, since the test runner would use the doc strings for reporting instead of the method names, which makes it much harder to find failing tests. Instead, choose an instructive method name – do not worry if it is long – and write detailed comments if necessary. Finally, if a test case tests a particular component, then you should always include an interface verification test:

```
def test_interface(self):
    self.assert_(IExample.providedBy(SimpleExample()))
```

This is probably the most trivial test you write, but also a very important one, since it tells you whether you fully implemented the interface.

5.2.6 Why is Having and Following the Conventions Important?

While I addressed some of the motivations for strict naming conventions at the beginning of the chapter, here are some more reasons. First off, there are many people working on the project. Everybody has an idea of an ideal style for his/her code, but in a project like this, only one can be adopted. If none would be created, people would constantly correct others' code and at the end we would have a most colorful mix of styles and the code would be literally unmaintainable.

Naming conventions also make the the API more predictable and are consequently more "usable" in a sense. For example, if I only remember the name of a class, like "simple example", I will always now how the spelling of the class will look like, i.e. `SimpleExample`. There is no ambiguity in the name at all.

A lot of time was spent on developing the directory structure. The goal is to have a well thought-out and flat tree so that the imports of the corresponding modules would stay manageable. Eventually we adopted the Java design for this reason. The development team also desired to be able to separate contract, implementation and presentation in order to address the different development roles (developer, scriptor, graphic designer, information architect and translator) of the individuals contributing to the project.

Ultimately we hope that these conventions are followed by all ZPL code in the `zope.org` source code repository.

For more information see the Zope 3 development wiki page at: `http://dev.zope.org/Zope3/CodingStyle`

CHAPTER 6

AN INTRODUCTION TO INTERFACES

Difficulty

Newcomer

Skills

- You should know Python well, which is a requirement for all chapters.
- Some knowledge about the usage of formal interfaces in software design. Optional.

Problem/Task

In every chapter in this book you will hear about interfaces in one way or another. Hence it is very important for the reader to understand the purpose of interfaces.

Solution

6.1 Introduction

In very large software projects, especially where the interaction with a lot of other software is expected and desired, it is necessary to develop well-specified application programming interfaces (APIs). We could think of APIs as standards of the framework, such as the RFC or POSIX standards. Once an interface is defined and made public, it should be very hard to change it. But an API is also useful inside a single software, known as internal API.

Interfaces (in the sense we will use the term) provide a programmatic way to specify the API in a programming language.

While other modern programming languages like Java use interfaces as a native language feature, Python did not even have the equivalent of a formal interface until recently. Usually, in Python the API is defined by the class that implements it, and it was up to the documentation to create a formal representation of the API. This approach has many issues. Often developers changed the API of a class without realizing that they broke many other people's code. Programmed interfaces can completely resolve this issue, since alarm bells can be rung (via tests) as soon as an API breakage occurs. Here is a simple example of a Python interface (as used by the Zope project):

```
1  from zope.interface import Interface, Attribute
2
3  class IExample(Interface):
4      """This interface represents a generic example."""
5
6      text = Attribute("The text of the example")
7
8      def setText(text):
9          "This method writes the passed text to the text attribute."
10
11     def getText():
12         "This method returns the value of the text attribute."
```

▷ Line 1: We import the only two important objects from `zope.interface` here, the meta-class `Interface` and the class `Attribute` .

▷ Line 3: We "misuse" the `class` declaration to declare interfaces as well. Note though, that interfaces are *not* classes and do not behave as such! Using `Interface` as base class will make this object an interface.

▷ Line 4: In Zope 3 interfaces are the main source for API documentation, so that it is necessary to always write very descriptive doc strings. The interface doc string gives a detailed explanation on how objects implementing this interface are expected to function.

▷ Line 6: The `Attribute` class is used to declare attributes in an interface. The constructor of this class only takes one string argument, which is the documentation of this attribute. You might say now that there is much more meta-data that an attribute could have, such as the minimum value of an integer or the maximum length of a string for example. This type of extensions to the `Attribute` class are provided by another module called `zope.schema`, which is described in "Zope Schemas and Widgets (Forms)".

▷ Line 8–9 & 11–12: Methods are declared using the `def` keyword as usual. The difference is, though, that the first argument is not `self`. You only list all of the common arguments. The doc string is again used for documentation.

Other than that, methods can contain anything you like; yet, Zope does not use anything else of the method content. If you use the `zope.interface` package apart from Zope 3, you could use the method body to provide formal descriptions of pre- and postconditions, argument types and return types.

The above is a typical but not practical example of an interface. Since we use Python, it is not necessary to specify both, the attribute and the setter/getter methods. In this case we would usually just specify the attribute as part of the interface and implement it as a Python property if necessary.

6.2 Advanced Uses

Once you have Python-based interfaces, many new possibilities develop. You can now use interfaces as objects that can define contracts. For example, you can say that there is a class `AllText` that converts `IExample` to `IAllText`, where the latter interface has a method `getAllText()` that returns all human-readable text from `IExample`. Such a class is known as an Adapter. More formally, adapters use one interface (`IExample`) to provide/implement another interface (`IAllText`).

Even more commonly, interfaces are used for identification. Zope 3's utility registry often executes queries of the form: "Give me all utilities that implement interface I1." Interfaces are even used to classify other interfaces! For example, I might declare my `IExample` interface to be an `IContentType`. I can then go to the utility registry and ask: "Give me all interfaces that represent a content type (`IContentType`)." Once I know these content type interfaces, I can figure out which classes are content types.

You can see that interfaces provide a solution to a wide range of use cases. By the way, it is very common to create empty interfaces purely for identification purposes; these interfaces are known as "marker interfaces".

6.3 Using Interfaces

In objects interfaces are used as follows:

```
1 from zope.interface import implements
2 from interfaces import IExample
3
4 class SimpleExample:
5     implements(IExample)
```

The `implements()` method tells the system that *instances* of the class provide `IExample`. But of course, modules and classes themselves can implement interfaces as well. For modules you can use `moduleProvides(*interfaces)`. For classes you can insert `classImplements(*interfaces)` directly in the class definition or use `classProvides(cls,*interfaces)` for an existing class. Also, you can use `directlyProvides(instance,*interfaces)` for any object as well (including instances of classes).

The `Interface` object itself has some nice methods. The most common one is `providedBy(ob)`, which checks whether the passed object implements the interface or not:

```
1 >>> ex = SimpleExample()
2 >>> IExample.providedBy(ex)
3 True
```

Similarly you can pass in a class and test whether instances of this class implement the interface by calling `IExample.implementedBy(SimpleExample)`.

The last useful method is `isOrExtends(interface)`. This method checks whether the passed interface equals the interface or whether the passed interface is a base (extends) of the interface.

When creating classes from an interface, there is a helpful script called `pyskel.py` that creates a class skeleton from an interface. Before using the script you have to make sure the `ZOPE3/src` is in your Python path. Usage:

```
python2.3 utilities/pyskel.py dotted.path.ref.to.interface
```

This call creates a skeleton of the class in your console, which saves you a lot of typing. The order of the attributes and methods as they appear in the interface is preserved.

As a last note, since Python does not comone with interfaces, the `zope.interface` package provides some interfaces that are implemented by the built-in Python types. They can be found in `zope.interface.common`.

This concludes the overview of interfaces. This introduction should be sufficient to get you started with the chapters. Many of the concepts will become much clearer as you work through the hands-on examples of the following parts of the book.

CHAPTER 7

THE COMPONENT ARCHITECTURE – AN INTRODUCTION

Difficulty

Newcomer

Skills

- Be familiar with object-oriented programming.
- Be knowledgeable about interfaces, i.e. by reading the previous chapter on interfaces.
- Knowledge of component-oriented programming is preferable. Optional.

Problem/Task

When the Component Architecture for Zope was first thought about, it was intended as an extension to Zope 2, not a replacement as it developed to become. The issue was that the existing Zope 2 API was too bloated and inconsistent, due to constant feature additions, bug fixes and coding inconsistencies. The extremely bad practice of "monkey patching" became a normality among developers to overcome the limitations of the API and fix bugs. Monkey patching is a method of overwriting library functions and class methods after importing them, which is a powerful, but dangerous, side effect of loosely-typed scripting languages.

Another motivation was to incorporate the lessons learned from consulting jobs and building large Web applications, which demonstrated the practical limits of simple object inheritance. The need for a more loosely-connected architecture arose

with many objects having rather tiny interfaces in contrast to Zope 2's few, large objects. This type of framework would also drastically reduce the learning curve, since a developer would need to learn fewer APIs to accomplish a given task.

All these requirements pointed to a component-oriented framework that is now known as the "Component Architecture" of Zope 3. Many large software projects have already turned to component-based systems. Some of the better-known projects include:

- COM (Microsoft's Object Model)
- Corba (Open source object communication protocol)
- KParts (from the KDE project)
- Mozilla API (it is some ways very similar to Zope 3's CA)
- JMX (Sun's Java Management Extensions manages Beans)

However, while Zope 3 has many similarities with the above architectures, thanks to Python certain flexibilities are possible that compiled languages do not allow.

Solution

In this chapter I will give you a high-level introduction to all component types. Throughout the book there will be concrete examples on developing most of the component types introduced here.

7.1 Services

Services provide fundamental functionality without which the application server would fail to function. They correspond to "Tools" in CMF (Zope 2) from which some of the semantics were also inspired.

Services do not depend on other components at all. You only interact with other components by passing them as arguments to the constructor or the methods of the service. Any given application should have only a few services that cover the most fundamental functionality. When dealing with locality, services should always delegate requests upward – up to the global version of the service – if a request can not be answered locally.

The most fundamental services are the registries of the components themselves. Whenever you register a class as a utility using ZCML, for example, then the class is registered in the "Utility Service" and can be retrieved later using the service. And yes, we also have a "Service Service" that manages all registered services.

Another service is the Error Reporting service, which records all errors that occurred during the publication process of a page. It allows a developer to review the details of the error and the state of the system/request at the time the error occurred.

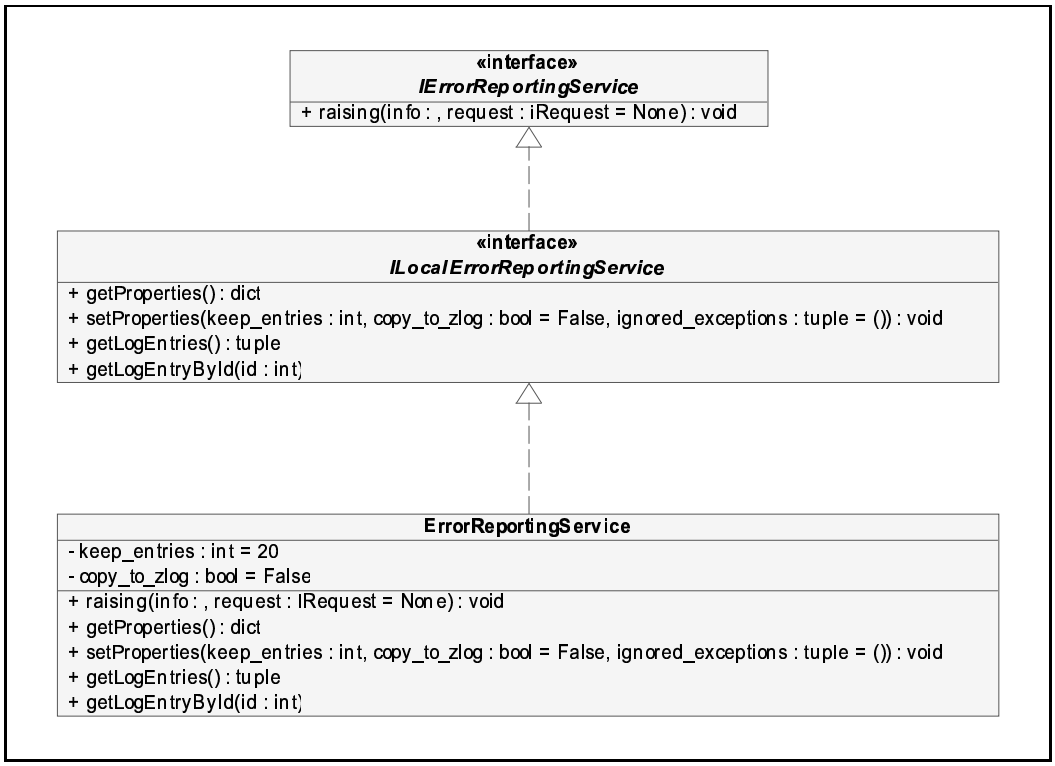


Figure 7.1: UML diagram of the Error Reporting Service.

A convention for service interfaces is that they only contain accessor methods. Mutator methods are usually implementation-specific and are provided by additional interfaces. A consequence of this pattern is that services are usually not modified once the system is running. Please note though that we strongly *discourage* developers from writing services for applications. Please use utilities instead.

Services make use of acquisition by delegating a query if it cannot be answered locally. For example, if I want to find a utility named "hello 1" providing the interface `IHello` and it cannot be found at my local site, then the Utility Service will delegate the request to the parent site. This goes all the way up to the global Utility Service. Only if the global service cannot provide a result, an error is returned. For more details about local and global components see "Global versus Local" below.

7.2 Adapters

Adapters could be considered the "glue components" of the architecture, since they connect one interface with another and allow various advanced programming techniques such as Aspect-Oriented Programming. An adapter takes a component implementing one interface and uses this object to provide another interface.

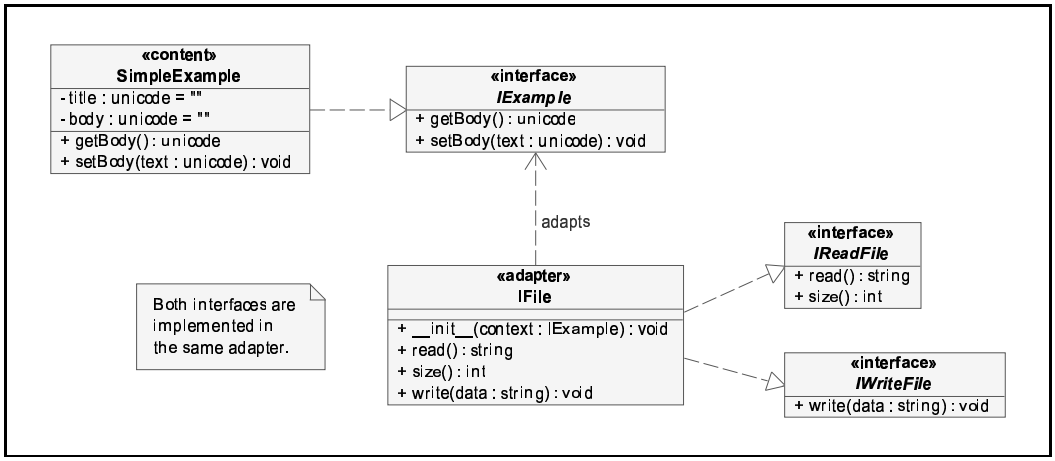


Figure 7.2: UML diagram of an Adapter adapting `IExample` to `IReadFile` and `IWriteFile`.

This allows the developer to split up the functionality into small API pieces and keep the functionality manageable. For example, one could write an adapter that allows an `IExample` content component to be represented as a file in FTP (see diagram above). This can be done by implementing the `IReadFile` and `IWriteFile` interface for the content component. Instead of adding this functionality directly to the `SimpleExample` class by implementing the interfaces in the class, we create an adapter that adapts `IExample` to `IReadFile` and `IWriteFile`. Once the adapter is registered for both interfaces (usually through ZCML), it can be used as follows:

```
1 read_file = zapi.getAdapter(example, IReadFile)
2 write_file = zapi.getAdapter(example, IWriteFile)
```

The `getAdapter()` method finds an adapter that maps any of the interfaces that are implemented by `example` (`SimpleExample` instance) to `IReadFile`. An optional argument named `context` can be passed as a keyword argument, specifying the place to look for the adapter. `None` causes the system to look only for global adapters. The default is the site the request was made from.

In this particular case we adapted from one interface to another. But adapters can also adapt from several interface to another. These are known as multi-adapters.

While multi-adapters were first thought of as unnecessary, they are now used in a wide range of applications.

The best side effect of adapters is that it was not necessary to touch the original implementation `SimpleExample` at all. This means that I can use any Python product in Zope 3 by integrating it using adapters and ZCML.

7.3 Utilities

Utilities are similar to services, but do not provide vital functionality, so applications should not be broken if utilities are missing. This statement should be clarified by an example.

In pre-alpha development of Zope 3, SQL Connections to various relational databases were managed by a service. The SQL Connection Service would manage SQL connections and the user could then ask the service for SQL connections by name. If a connection was not available, then the service would give a negative answer. Then we realized the role of utilities, and we were able to rid ourselves of the SQL Connection Service and implement SQL connections as utilities. Now we can ask the Utility Service to give us an object implementing `ISQLConnection` and having a specified name. We realized that many services that merely acted as registries could be thrown out and the objects they managed became utilities. This greatly reduced the number of services and the complexity of the system. The lesson here is that before you develop a service, evaluate whether it would just act as a container, in which case the functionality is better implemented using utilities.

7.4 Factories (Object Classes/Prototypes)

Factories, as the name suggests, exist merely to create other components and objects. Factories can be methods, classes or even instances that are callable. The developer only encounters them directly when dealing with content objects (since ZCML creates factories for you automatically) if you specify the `factory` directive. The functionality and usefulness of factories is best described by an example.

Let's consider our `SimpleExample` content component once more. A factory has to provide two methods. The obvious one is the `__call__()` method that creates and returns a `SimpleExample` instance. The second method, called `getInterfaces()` returns a list of interfaces that the object created by calling `__call__` will provide.

Factories are simply named utilities that provide `IFactory`. Using a special sub-directive you can register a factory using an id, such as `example.SimpleExample`. Once the factory is registered, you can use the component ar-

chitecture's `createObject()` method to create Simple Examples using the factory (implicitly):

```
1 ex = zapi.createObject('example.SimpleExample')
```

The argument is simply the factory id. By the way, a factory id must be unique in the entire system and the low-level functionality of factories is mostly hidden by high-level configuration. Optionally you can specify a context argument, that specifies the location you are looking in. By default it is the site of the request; if you specify `None`, only global factories are considered.

7.5 Presentation Components – Views, Resources, Skins, Layers

Presentation components, especially views, are very similar to adapters, except that they take additional parameters like layers and skins into account. In fact, in future versions of Zope 3, the presentation service will be removed and presentation components will become adapters. Presentation components are used for providing presentation for other components in various output formats (presentation types), such as HTML, FTP, XML-RPC and so on.

In order to make a *view* work, two pieces of information have to be provided. First, the view must know for which object it is providing a view. This object is commonly known as the `context` of the view. Second, we need to know some protocol-specific information, which is stored in a `Request` object that is always accessible under the variable name `request` in the view. For HTML, for example, the request contains all cookies, form variables and HTTP header values, but also the authenticated user and the applicable locale. The return values of the methods of a view depend on the presentation type and the method itself. For example, HTTP views usually return the HTML body, whereas FTP might return a list of "filenames".

Resources are presentation components in their own right. In comparison to views, they do not provide a presentation of another component but provide some presentation output that is independent of any other component. HTML is a prime example. Stylesheets and images (layout) are often not considered content and also do not depend on any content object, yet they are presentation for the HTTP presentation type. However, not all presentation types require resources; both FTP and XML-RPC do not have such independent presentation components.

Next, views and resources are grouped by layers, which are usually used for grouping similar look and feel. In Zope 2's CMF framework, layers are the folders contained by the `portal_skins` tool. An example for a layer is `debug`, which simply inserts Python tracebacks into exception HTML pages.

Multiple layers can then be stacked together to skins. Currently we have several skins in the Zope 3 core: "rotterdam" (default), "Basic", "Debug", and "ZopeTop" (excluded from the 3.0 distribution). The skin can simply be changed by typing `++skin++SKINNAME` after the root of your URL, for example:

`http://localhost:8080/++skin++ZopeTop/folder1`

When you develop an end-user Web site, you definitely want to create your own layer and incorporate it as a new skin. You want to avoid writing views for your site that each enforce the look and feel. Instead you can use skins to create a look and feel that will work for all new and existing templates.

7.6 Global versus Local

Zope 3 separates consciously between local and global components. Global components have no place associated with them and are therefore always reachable and present. They are always initialized and registered at Zope 3 startup via the Zope Configuration Markup Language (ZCML), which I mentioned before. Therefore, global components are *not* persistent meaning they are not stored in the ZODB at all. Their state is destroyed (and should be) upon server shutdown.

Local components, on the other hand, are stored in the ZODB at the place in the object tree they were defined in. Local components always only add and overwrite previous settings; they can never remove existing ones. Creating new local components can be done using the Web Interface by clicking "Manage Site". This will lead you into the configuration namespace and is always marked by `++etc++site` in the URL.

Every folder can be promoted to become a site. Once a local site manager is declared for a folder by clicking on "Make Site", we call this object/folder a site.

As mentioned before, local components use an explicit acquisition process when looking up information. For example, I want to get the factory for `SimpleExample`.

When looking for any component, the original site of the publication is chosen. However, sometimes it is desired to start looking for a component from a different site. In these cases you simply specify the `context` in the call.

```
1 factory = zapi.getFactory('example.SimpleExample', context=other_site)
```

If a local Utility Service exists and an `IFactory` utility with the name `example.SimpleExample` is found, then it is returned. If not, then the local Utility Service delegates the request to the next site. Requests can be delegated all the way up to the global Utility Service at which point an answer must be given. If the global Utility Service does not know the answer either, a `ComponentLookupError` is raised.

We can see that there are slight semantic differences between global and local implementations of a service, besides the difference in data storage and accessibility. The global service never has to worry about place or the delegation of the request. The net effect is that global components are often easier to implement than their local equivalent. Furthermore, local components usually have writable APIs in addition to the readable ones, since they have to allow run-time management.

Chapter 8

Zope Schemas and Widgets (Forms)

Difficulty

Newcomer

Skills

- Be familiar with the various built-in Python types.
- Be familiar with HTML forms and CGI.
- Knowledge about Formulator (Zope 2) is of advantage. Optional.

Problem/Task

In the early stages of development, the Zope 3 developers decided that it would be cumbersome to manually write HTML forms and to manually validate the input. We realized that if we would extend interfaces, we could auto-generate HTML forms and also automatically validate any input. This chapter gives some background information and formally introduces the `zope.schema` and `zope.app.form` packages.

Solution

8.1 History and Motivation

Originally, I simply wanted to port Formulator, a very successful Zope 2 product to auto-generate and validate forms, to Zope 3. In Formulator, one would create various input fields (like integers or text lines) in a form and provide some meta-data about the fields, like the maximum and minimum length of a string. You could then tell the form to simply render itself. For more details see `http://zope.org/Members/infrae/Formulator`.

Even though Formulator tried to split application logic and presentation, various parts were still not sufficiently separated, mainly due to the limitations Zope 2 provided. Therefore the original port remained a hack in Zope 3 until the idea of schemas was developed by Jim Fulton and Martijn Faassen (the original author of Formulator) during the Berlin BBQ Sprint (April 2002) when trying to combine Zope 3's version of Formulator and class properties. After all presentation logic was removed, Formulator fields felt a lot like interface specification for attributes. So it was realized, that if we supply more meta-data to the attribute declarations in interfaces, then we could accomplish auto-generation and validation of HTML forms. These extended attributes are still known as "fields". If an interface contains any fields, then this interface is conventionally called a schema.

The following three main goals of schemas developed:

1. Full specification of properties on an API level
2. Data input validation and conversion
3. Automated GUI form generation (mainly for the Web browser)

8.2 Schema versus Interfaces

As mentioned before, schemas are just an extension to interfaces and therefore depend on the `zope.interface` package. Fields in schemas are equivalent to methods in interfaces. Both are complementary to each other, since they describe different aspects of an object. The methods of an interface describe the functionality of a component, while the schema's fields represent the state.

It is thus not necessary to develop a new syntax for writing schemas and we simply reuse the interface declaration:

```
1 from zope.interface import Interface
2 from zope.schema import Text
3
4 class IExample(Interface):
5
6     text = Text(
7         title=u"Text",
```

```
8         description=u"The text of the example.",
9         required=True)
```

▷ Line 2: All default fields can be simply imported from `zope.schema`.

▷ Line 7–8: The title and description are used as human-readable text for the form generation. Of course, they also serve as documentation of the field itself.

▷ Line 9: Various fields support several other meta-data fields. The `required` option is actually available for all fields and specifies whether an object implementing `IExample` must provide a value for `text` or not.

8.3 Core Schema Fields

After we have seen a simple example of a schema, let's now look at all the basic fields and their properties.

- Properties that all fields support:
  - `title` (type: `TextLine`): The title of the attribute is used as label when displaying the field widget.
  - `description` (type: `Text`): The description of the attribute is used for tooltips and advanced help.
  - `required` (type: `Bool`): Specifies whether an attribute is required or not to have a value. In add-forms, required attributes are equivalent to required constructor arguments.
  - `readonly` (type: `Bool`): If a field is readonly, then the value of the attribute can be set only once and can then only be displayed. Often a unique id for some object is a good candidate for a read-only field.
  - `default` (type: depends on field): The default value that is given to the attribute, if no initialization value was provided. This value is often specified, if a field is required.
  - `order` (type: `Int`): Fields are often grouped by some logical order. This value specifies a relative position in this order. We usually do not set this value manually, since it is automatically assigned when an interface is initialized. The order of the fields in a schema is by default the same as the order of the fields in the Python code.
- `Bytes`, `BytesLine`

`Bytes` and `BytesLine` only differ by the fact that `BytesLine` cannot contain a new line character. `Bytes` behave identical to the Python type `str`.

`Bytes` and `BytesLine` fields are iteratable.

  - `min_length` (type: `Int`): After the white space has been normalized, there cannot be less than this amount of characters in the bytes string. The default is `None`, which refers to no minimum.
  - `max_length` (type: `Int`): After the white space has been normalized, there cannot be more than this amount of characters in the bytes string. The default is `None`, which refers to no maximum.

- `Text`, `TextLine`

  The two fields only differ by the fact that `TextLine` cannot contain a newline character. `Text` fields contain unicode, meaning that they are intended to be human-readable strings/text.

  `Text` and `TextLine` fields are iteratable.

  - `min_length` (type: `Int`): After the white space has been normalized, there cannot be less than this amount of characters in the text string. The default is `None`, which refers to no minimum.
  - `max_length` (type: `Int`): After the white space has been normalized, there cannot be more than this amount of characters in the text string. The default is `None`, which refers to no maximum.

- `SourceText`

  Source Text is a special field derived from `Text`, which contains source code of any type. It is more or less a marker field for the forms machinery, so that special input fields can be used for source code.

- `Password`

  `Password` is a special derivative for the `TextLine` field and is treated separately for presentation reasons. However, someone also might want more fine-grained validation for passwords.

- `Bool`

  The `Bool` field has no further attributes. It maps directly to Python's `bool` object.

- `Int`

  `Int` fields directly map to Python's `int` type.

  - `min` (type: `Int`): Specifies the smallest acceptable integer. This is useful in many ways, such as allowing only positive values by making this field `0`.
  - `max` (type: `Int`): Specifies the largest acceptable integer, which excludes the value itself. It can be used to specify an upper bound, such as the current year, if you are interested in the past only.

  Both attributes combined allow the programmer to specify ranges of acceptable values.

- `Float`

  `Float` fields directly map to Python's `float` type.

  - `min` (type: `Float`): Specifies the smallest acceptable floating point number. This is useful in many ways, such as allowing only positive values by making this field `0.0`.
  - `max` (type: `Float`): Specifies the largest acceptable floating point number, which excludes the value itself (typical computer programming pattern). It can be used to specify an upper bound, such as `1.0`, if you are only interested in probabilities.

  Both attributes combined allow the programmer to specify ranges of acceptable values.

- `Datetime`

  Similar to `Int` and `Float`, `Datetime` has a `min` and `max` field that specify the boundaries of the possible values. Acceptable values for these fields must be instances of the builtin `datetime` type.

- `Tuple`, `List`

  The reason both of these fields exists is that we can easily map them to their Python type `tuple` and `list`, respectively.

  `Tuple` and `List` fields are iteratable.

  - `min_length` (type: `Int`): There cannot be less than this amount of items in the sequence. The default is `None`, which means there is no minimum.
  - `max_length` (type: `Int`): There cannot be more than this amount of items in the sequence. The default is `None`, which means there is no maximum.
  - `value_type` (type: `Field`): Values contained by these sequence types must conform to this field's constraint. Most commonly a `Choice` field (see below) is specified here, which allows you to select from a fixed set of values.

- `Dict`

  The `Dict` is a mapping field that maps from one set of fields to another.

  `Dict` fields are iteratable.

  - `min_length` (type: `Int`): There cannot be less than this amount of items in the dictionary. The default is `None`, which means there is no minimum.
  - `max_length` (type: `Int`): There cannot be more than this amount of items in the dictionary. The default is `None`, which means there is no maximum.
  - `key_type` (type: `Field`): Every dictionary item key has to conform to the specified field.
  - `value_type` (type: `Field`): Every dictionary item value has to conform to the specified field.

- `Choice`

  The `Choice` field allows one to select a particular value from a provided set of values. You can either provide the values as a simple sequence (list or tuple) or specify a vocabulary (by reference or name) that will provide the values. Vocabularies provide a flexible list of values, in other words the set of allowed values can change as the system changes. Since they are so complex, they are covered separately in "Vocabularies and Fields".

  - `vocabulary` (type: `Vocabulary`): A vocabulary instance that is used to provide the available values. This attribute is `None`, if a vocabulary name was specified and the field has not been bound to a context.
  - `vocabularyName` (type: `TextLine`): The name of the vocabulary that is used to provide the values. The vocabulary for this name can only be looked up, when the field is bound, in other words has a context. Upon binding, the vocabulary is automatically looked using the name and the context.

  The constructor also accepts a `values` argument that specifies a static set of values. These values are immediately converted to a static vocabulary.

- `Object`

  This field specifies an object that must implement a specific schema. Only objects that provide the specified schema are allowed.

  - `schema` (type: `Interface`): This field provides a reference to the schema that must be provided by objects that want to be stored in the described attribute.

- `DottedName`

  Derived from the `BytesLine` field, the `DottedName` field represents valid Python-style dotted names (object references). This field can be used when it is desirable that a valid and resolvable Python dotted name is provided.

  This field has no further attributes.

- `URI`

  Derived from the `BytesLine` field, the `URI` field makes sure that the value is always a valid URI. This is particularly useful when you want to reference resources (such as RSS feeds or images) on remote computers.

  This field has no further attributes.

- `Id`

  Both, the `DottedName` and `URI` field, make up the `Id` field. Any dotted name or URI represent a valid id in Zope. Ids are used for identifying many types of objects, such as permissions and principals, but also for providing annotation keys.

  This field has no further attributes.

- `InterfaceField`

  The `Interface` field has no further attributes. Its value must be an object that provides `zope.interface.Interface`, in other words it must be an interface.

For a formal listing of the Schema/Field API, see the API documentation tool at `http://localhost:8080/++apidoc++` or see `zope.schema.interfaces` module.

8.4 Auto-generated Forms using the `forms` Package

Forms are much more Zope-specific than schemas and can be found in the `zope.app.forms` package. The views of schema fields are called widgets. Widgets responsible for data display and conversion in their specific presentation type. Currently widgets exist mainly for HTML (the Web browser).

Widgets are separated into two groups, display and input widgets. Display widgets are often very simply and only show a text representation of the Python object. The input widgets, however, are more complex and display a greater variety of choices. The following list shows all available browser-based input widgets (found in `zope.app.form.browser`):

Text Widgets

Text–based widgets always require some sort of keyboard input. A string representation of a field is then converted to the desired Python object, like and integer or a date.

- `TextWidget`: Being probably the simplest widget, it displays the `text` input element and is mainly used for the `TextLine`, which expects to be unicode. It also serves as base widget for many of the following widgets.
- `TextAreaWidget`: As the name suggests this widget displays a text area and assumes its input to be some unicode string. (note that the Publisher already takes care of the encoding issues).
- `BytesWidget`, `BytesAreaWidget`: Direct descendents from `TextWidget` and `TextAreaWidget`, the only difference is that these widgets expect bytes as input and not a unicode string, which means they must be valid ASCII encodable.
- `ASCIIWidget`: This widget, based on the `BytesWidget`, ensures that only ASCII character are part of the inputted data.
- `PasswordWidget`: Almost identical to the `TextWidget`, it only displays a `password` element instead of a `text` element.
- `IntWidget`: A derivative of `TextWidget`, it only overwrites the conversion method to ensure the conversion to an integer.
- `FloatWidget`: Derivative of `TextWidget`, it only overwrites the conversion method to ensure the conversion to an floating point.
- `DatetimeWidget`: Someone might expect a smart and complex widget at this point, but for now it is just a simple `TextWidget` with a string to `datetime` converter. There is also a `DateWidget` that only handles dates.

Boolean Widgets

Boolean widgets' only responsibility is to convert some binary input to the Python values `True` or `False`.

- `CheckBoxWidget`: This widget displays a single checkbox widget that can be either checked or unchecked, representing the state of the boolean value.
- `BooleanRadioWidget`: Two radio buttons are used to represent the true and false state of the boolean. One can pass the textual value for the two states in the constructor. The default is "on" and "off" (or their translation for languages other than English).

- `BooleanSelectWidget`, `BooleanDropdownWidget`: Similar to the `BooleanRadioWidget`, textual representations of the true and false state are used to select the value. See `SelectWidget` and `DropdownWidget`, respectively, for more details.

Single Selection Widgets

Widgets that allow a single item to be selected from a list of values are usually views of a field, a vocabulary and the request, instead of just the field and request pair. Therefore so called proxy–widgets are used to map from field–request to field–vocabulary–request pairs. For example the `ChoiceInputWidget`, which takes a `Choice` field and a request object, is simply a function that looks up another widget that is registered for the `Choice` field, its vocabulary and the request. Below is a list of all available widgets that require the latter three inputs.

- `SelectWidget`: This widget provides a multiply–sized selection element where the options are populated through the vocabulary terms. If the field is not required, a "no value" option will be available as well. The user will allowed to only select one value though, since the `Choice` field is not a sequence–based field.

- `DropdownWidget`: As a simple derivative of the `SelectWdiget`, it has its size set to "1", which makes it a dropdown box. Dropdown boxes have the advantage that they always just show one value, which makes some more user–friendly for single selections.

- `RadioWidget`: This widget displays a radio button for each term in the vocabulary. Radio buttons have the advantage that they always show all choices and are therefore well suitable for small vocabularies.

Multiple Selections Widgets

This group of widgets is used to display input forms collection–based fields, such as `List` or `Set`. Similar to the single selection widgets, two proxy-widgets are used to look up the correct widget. The first step is to map from `field– request` to `field– value_type– request` using a widget called `CollectionInputWidget`. This allows us to use different widgets when the value type is an `Int` or `Choice` field for example. Optionally, a second proxy–widget is used to convert the `field– value_type– request` pair to a `field– vocabulary– request` pair, as it is the case when the value type is a choice field.

- `MultiSelectWidget`: Creates a select element with the `multiple` attribute set to true. This creates a multi-selection box. This is especially useful for vocabularies with many terms. Note that if your vocabulary supports a query interface, you can even filter your selectable items using queries.

- `MultiCheckBoxWidget`: Similar to the multi-selection widget, this widget allows multi-value selections of a given list, but uses checkboxes instead of a list. This widget is more useful for smaller vocabularies.
- `TupleSequenceWidget`: This widget is used for all cases where the value type is not a `Choice` field. It used the widget of the value type field to add new values to the tuple. Other input elements are used to remove items.
- `ListSequenceWidget`: This widget is equivalent to the previous one, except that it generates lists instead of tuples.

Miscellaneous Widgets

- `FileWidget`: This widget displays a file input element and makes sure the received data is a file. This field is ideal for quickly uploading byte streams as required for the `Bytes` field.
- `ObjectWidget`: The `ObjectWidget` is the view for an object field. It uses the schema of the object to construct an input form. The object factory, which is passed in as a constructor argument, is used to build the object from the input afterwards.

Here is a simple interactive example demonstrating the rendering and conversion functionality of a widget:

```
1  >>> from zope.publisher.browser import TestRequest
2  >>> from zope.schema import Int
3  >>> from zope.app.form.browser import IntWidget
4  >>> field = Int(__name__='number', title=u'Number', min=0, max=10)
5  >>> request = TestRequest(form={'field.number': u'9'})
6  >>> widget = IntWidget(field, request)
7  >>> widget.hasInput()
8  True
9  >>> widget.getInputValue()
10 9
11 >>> print widget().replace(' ', '\n  ')
12 <input
13   class="textType"
14   id="field.number"
15   name="field.number"
16   size="10"
17   type="text"
18   value="9"
19
20   />
```

▷ Line 1 & 5: For views, including widgets, we always need a request object. The `TestRequest` class is the quick and easy way to create a request without much hassle. For each presentation type there exists a `TestRequest` class. The class

takes a `form` argument, which is a dictionary of values contained in the HTML form. The widget will later access this information.

- Line 2: Import an integer field.
- Line 3 & 6: Import the widget that displays and converts an integer from the HTML form. Initializing a widget only requires a field and a request.
- Line 4: Create an integer field with the constraint that the value must lie between 0 and 10. The `__name__` argument must be passed here, since the field has not been initialized inside an interface, where the `__name__` would be automatically assigned.
- Line 7–8: This method checks whether the form contained a value for this widget.
- Line 9–10: If so, then we can use the `getInputValue()` method to return the converted and validated value (an integer in this case). If we would have chosen an integer outside this range, a `WidgetInputError` would have been raised.
- Line 11–20: Display the HTML representation of the widget. The `replace()` call is only for better readability of the output.

Note that you usually will not have to deal with these methods at all manually, since the form generator and data converter does all the work for you. The only method you will commonly overwrite is `_validate()`, which you will use to validate custom values. This brings us right into the next subject, customizing widgets.

There are two ways of customizing widgets. For small adjustments to some parameters (properties of the widget), one can use the `browser:widget` subdirective of the `browser:addform` and `browser:editform` directives. For example, to change the widget for a field called "name", the following ZCML code can be used.

```
1  <browser:addform
2    ... >
3
4    <browser:widget
5        field="name"
6        class="zope.app.form.browser.TextWidget"
7        displayWidth="45"
8        style="width: 100%"/>
9
10 </browser:addform>
```

In this case we force the system to use the `TextWidget` for the name, set the display width to 45 characters and add a style attribute that should try to set the width of the input box to the available width.

The second possibility to change the widget of a field is to write a custom view class. In there, custom widgets are easily realized using the `CustomWidget` wrapper class. Here is a brief example:

```
1 from zope.app.form.widget import CustomWidget
2 from zope.app.form.browser import TextWidget
3
4 class CustomTextWidget(TextWidget):
5     ...
6
7 class SomeView:
8     name_widget = CustomWidget(CustomTextWidget)
```

- ▷ Line 1: Since `CustomWidget` is presentation type independent, it is defined in `zope.app.form.widget`.
- ▷ Line 4–5: You simply extend an existing widget. Here you can overwrite everything, including the `_validate()` method.
- ▷ Line 7–8: You can hook in the custom widget by adding an attribute called `name_widget`, where `name` is the name of the field. The value of the attribute is a `CustomWidget` instance. `CustomWidget` has only one required constructor argument, which is the custom widget for the field. Other keyword arguments can be specified as well, which will be set as attributes on the widget.

More information about schemas can be found in the `README.txt` file of the `zope.schema` package. The Zope 3 development Web site also contains some additional material.

This concludes our introduction to schemas and forms. For examples of schemas and forms in practice, see the first chapters of the "Content Components – The Basics" part.

CHAPTER 9

INTRODUCTION TO THE ZOPE CONFIGURATION MARKUP LANGUAGE (ZCML)

Difficulty

Newcomer

Skills

- Be familiar with the previous chapters of this section, specifically the introduction to the component architecture.
- Some basic familiarity with XML is of advantage. Optional.

Problem/Task

Developing components alone does not make a framework. There must be some configuration utility that tells the system how the components work together to create the application server framework. This is done using the Zope Configuration Markup Language (ZCML) for all filesystem-based code. Therefore it is very important that a developer knows how to use ZCML to hook up his/her components to the framework.

Solution

As stated above, it became necessary to develop a method to setup and configure the components that make up the application server. While it might seem otherwise, it is not that easy to develop an effective configuration system, since there are

several requirements that must be satisfied. Over time the following high-level requirements developed that caused revisions of the implementation and coding styles to be created:

1. While the developer is certainly the one that writes the initial cut of the configuration, this user is not the real target audience. Once the product is written, you would expect a system administrator to interact much more frequently with the configuration, adding and removing functionality or adjust the configuration of the server setup. System administrators are often not developers, so that it would be unfortunate to write the configuration in the programming language, here Python. But an administrator is familiar with configuration scripts, shell code and XML to some extend. Therefore an easy to read syntax that is similar to other configuration files is of advantage.

2. Since the configuration is not written in Python, it is very important that the tight integration with Python is given. For example, it must be very simple to refer to the components in the Python modules and to internationalize any human-readable strings.

3. The configuration mechanism should be declarative and not provide any facilities for logical operations. If the configuration would support logic, it would become very hard to read and the initial state of the entire system would be unclear. This is another reason Python was not suited for this task.

4. Developing new components sometimes requires to extend the configuration mechanism. So it must be easy for the developer to extend the configuration mechanism without much hassle.

To satisfy the first requirement, we decided to use an XML-based language (as the name already suggests). The advantage of XML is also that it is a "standard format", which increases the likelihood for people to be able to read it right away. Furthermore, we can use standard Python modules to parse the format and XML namespaces help us to group the configuration by functionality.

A single configuration step is called a directive . Each directive is an XML tag, and therefore the tags are grouped by namespaces. Directives are done either by simple or complex directives. Complex directives can contain other sub-directives. They are usually used to provide a set of common properties, but do not generate an action immediately.

A typical configuration file would be:

```
1 <configure
2     xmlns="http://namespaces.zope.org/zope">
3
4 <adapter
```

```
5     factory="product.FromIXToIY"
6     for="product.interfaces.IX"
7     provides="product.interfaces.IY" />
8
9 </configure>
```

All configuration files are wrapped by the `configure` tag, which represents the beginning of the configuration. In the opening of this tag, we always list the namespaces we wish to use in this configuration file. Here we only want to use the generic Zope 3 namespace, which is used as the default. Then we register an adapter with the system on line 4–7. The interfaces and classes are referred to by a proper Python dotted name. The `configure` tag might also contain an `i18n_domain` attribute that contains the domain that is used for all the translatable strings in the configuration.

As everywhere in Zope 3, there are several naming and coding conventions for ZCML inside a package. By default you should name the configuration file `configure.zcml`. Inside the file you should only declare namespaces that you are actually going to use. When writing the directives make sure to logically group directives together and use comments as necessary. Comments are written using the common XML syntax: ⟨`!--...--`⟩. For more info see Steve's detailed ZCML Style Guide at `http://dev.zope.org/Zope3/ZCMLStyleGuide` for more info.

To satisfy our fourth requirement, it is possible to easily extend ZCML through itself using the `meta` namespace . A directive can be completely described by four components, its name, the namespace it belongs to, the schema and the directive handler:

```
1 <meta:directive
2     namespace="http://namespaces.zope.org/zope"
3     name="adapter"
4     schema=".metadirectives.IAdapterDirective"
5     handler=".metaconfigure.adapterDirective" />
```

These meta-directives are commonly placed in a file called `meta.zcml`.

The schema of a directive, which commonly lives in a file called `metadirectives.py`, is a simple Zope 3 schema whose fields describe the available attributes for the directive. The configuration system uses the fields to convert and validate the values of the configuration for use. For example, dotted names are automatically converted to Python objects. There are several specialized fields specifically for the configuration machinery:

- `PythonIndentifier` – This field describes a python identifier, for example a simple variable name.
- `GlobalObject` – An object that can be accessed as a module global, such as a class, function or constant.

- `Tokens` – A sequence that can be read from a space-separated string. The `value_type` of the field describes token type.
- `Path` – A file path name, which may be input as a relative path. Input paths are converted to absolute paths and normalized.
- `Bool` – An extended boolean value. Values may be input (in upper or lower case) as any of: yes, no, y, n, true, false, t, or f.
- `MessageID` – Text string that should be translated. Therefore the directive schema is the only place that needs to deal with internationalization. This satisfies part of requirement 2 above.

The handler, which commonly lives in a file called `metaconfigure.py`, is a function or another callable object that knows what needs to be done with the given information of the directive. Here is a simple (simplified to the actual code) example:

```
def adapter(_context, factory, provides, for_, name=''):

    _context.action(
        discriminator = ('adapter', for_, provides, name),
        callable = provideAdapter,
        args = (for_, provides, factory, name),
        )
```

The first argument of the handler is always the `_context` variable, which has a similar function to `self` in classes. It provides some common methods necessary for handling directives. The following arguments are the attributes of the directive (and their names must match). If an attribute name equals a Python keyword, like `for` in the example, then an underscore is appended to the attribute name.

The handler should also not directly execute an action, since the system should first go through all the configuration and detect possible conflicts and overrides. Therefore the `_context` object has a method called `action` that registers an action to be executed at the end of the configuration process. The first argument is the `discriminator`, which uniquely defines a specific directive. The `callable` is the function that is executed to provoke the action, the `args` argument is a list of arguments that is passed to the callable and the `kw` contains the callable's keywords.

As you can see, there is nothing inheritly difficult about ZCML. Still, people coming to Zope 3 often experience ZCML as the most difficult part to understand. This often created huge discussions about the format of ZCML. However, I believe that the problem lies not within ZCML itself, but the task it tries to accomplish. The components themselves always seem so clean in implementation; and then you get to the configuration. There you have to register this adapter and that view, make security assertions, and so on. And this in itself seems overwhelming at first sight. When I look at a configuration file after a long time I often have this feeling

too, but reading directive for directive often helps me to get a quick overview of the functionality of the package. In fact, the configuration files can help you understand the processes of the Zope 3 framework without reading the code, since all of the interesting interactions are defined right there.

Furthermore, ZCML is well documented at many places, including the Zope 3 API documentation tool at `http://localhost:8080/++apidoc++/`. Here is a short list of the most important namespaces:

- `zope` – This is the most generic and fundamental namespace of all, since it allows you to register all basic components with the component architecture.
- `browser` – This namespace contains all of the directives that deal with HTML output, including managing skins and layer, declare new views (pages) and resources as well as setup auto-generated forms.
- `meta` – As discussed above, you can use this namespace to extend ZCML's available directives.
- `xmlrpc` – This is the equivalent to `browser`, except that allows one to specify methods of components that should be available via XML-RPC.
- `i18n` – This namespace contains all internationalization- and localization-specific configuration. Using `registerTranslations` you can register new message catalogs with a translation domain.
- `help` – Using the `register` directive, you can register new help pages with the help system. This will give you context-sensitive help for the ZMI screens of your products.
- `mail` – Using the directives of this namespace you can setup mailing components that your application can use to send out E-mails.

CHAPTER 10

INTRODUCTION TO ZOPE'S I18N AND L10N SUPPORT

Difficulty

Newcomer

Skills

- Be familiar with the previous chapters of this section, specifically the introduction to components.
- You should be familiar with common tasks and problems that arise when developing translatable software.
- Know about the gettext and/or ICU tools. Optional.

Problem/Task

Often it is not acceptable to provide an application in just one language and it must be possible to provide the software in many languages. But the problem is not solved there. Besides simple text, one must also handle date/time and number formats for example, since they are specific to regions and languages as well. This chapter will give the reader an overview of the utilities that Zope 3 provides to solve these issues.

Solution

10.1 History

One of the most severe issues of Zope 2 was the lack of multi-language support. This significantly limited the adoption of Zope outside English-speaking regions. Later support was partially added through add-on products like Localizer, ZBabel, which allowed translation of DTML and Python code (and therefore ZPT). However, these solutions could not overcome the great limitation that Zope 2 is not unicode aware. Several workarounds to the problem were provided, but they did not provide a solid solution.

Once the internationalization effort was initiated and the `i18n` Page Template namespace was developed for Zope 3, it was backported to Zope 2 and a Placeless Translation Service product was provided by the community (`http://www.zope.org/Members/efge/TranslationService`). [1]

When the Zope 3 development was opened to the community, it was realized that internationalization is one of the most important features, since Zope has a large market in Latin America, Asia and especially Europe. Therefore, the first public Zope 3 sprint in January 2002 was dedicated to this subject. Furthermore, Infrae paid me for two weeks to work on Zope 3's internationalization and localization support. Since then I have maintained and updated the internationalization and localization support for Zope 3.

10.2 Introduction

In the previous section I used the terms internationalization and localization, but what do they mean? Internationalization, often abbreviated as I18n, is the process to make a software translatable. This includes preparing and marking strings for translation, provide utilities to represent data (like dates/times and numbers) in regional formats and to be able to recognize the region/language setting of the user. The last section of this chapter will deal in detail on how to internationalize the various components of your Zope 3 code. Localization, on the other hand, is the process to translate the software to a particular language/region. For this task, one needs a tool to extract all translatable strings and another one to aid the translation process. Localization data for number formatting, currencies, timezones and much more are luckily already compiled in large volumes of XML-locale files.

There are three goals which the Zope 3 I18n support tries to accomplish:

1. The support will only deal with the translation of software, not content. Internationalizing and localizing content requires very custom software that implements very specific workflows.

[1] Zope 3 uses now Translation Domain Utilities instead of Translation Services.

2. Since Zope 3 is a network application server, instead of a simple application, the I18n solution should be flexible enough to support changing locale settings among different users. This is appreciably more difficult to implement than the I18n for an application that runs on a client.

3. It should be very simple and transparent to internationalize 3rd party add-on products.

In the Open Source world, there are two established solutions for providing I18n libraries and L10n utilities, GNU Gettext and ICU . The latter was primarily developed to replace the original Java I18n support. However, Gettext is the defacto standard for the Free Software world (for example KDE and Gnome), but it has some major shortcomings. Gettext only does the translation of messages (human readable strings) okay – not even well. On the other hand, there are many translation tools that support the gettext format, such as KBabel, a true power tool for translating message catalogs . Therefore, it is important to support the gettext message catalog format, even if it is only through import and export facilities.

ICU, in contrast, is a very extensive and well-developed framework that builds upon the experience of the Java I18n libraries. ICU provides objects for everything that you could ever imagine, including locales, object formatting and transliteration rules. The best of all is that the information of over 220 locales is available in XML files. These files contain complete translations of all countries and languages, date/-time formatting/parsing rules for three different formats (follow standard specification) – including all month/weekday names/abbreviations, timezone specifications (city names inclusive) – and number formatting/parsing rules for decimal, scientific, monetary, percent and per-mille numbers.

The first decision we made concerning I18n was to make all human-readable text unicode, so that we would not run into the same issues as Zope 2. Only the publisher would convert the unicode to ASCII (using UTF-8 or other encodings). The discussion and decision of this subject are immortalized in the proposal at `http://dev.zope.org/Zope3/UnicodeForText`).

Since the ICU framework is simply too massive to be ported to Python for Zope 3, we decided to adopt the locales support from ICU (using the XML files as data) and support the gettext message catalogs for translation, simply because the gettext tools are available as standard libraries in Python. From the XML locale files we mainly use the date/time and number patterns for formatting and parsing these data types. Two generic pattern parsing classes have been written respectively and can be used independently of Zope 3's I18n framework. On top of these pattern parsing classes are the formatter and parser class for each corresponding data type. But all this is hidden behind the Locale object, which makes all of the locale data available and provides some convenience functions.

10.3 Locales

The `Locale` instance for a user is available via the `request` object, which is always available from a view. However, one can easily test the functionality of Locale instances using the interactive Python prompt. Go to the directory `ZOPE3/src` and start Python. You can now use the following code to get a locale:

```
>>> from zope.i18n.locales import LocaleProvider
>>> provider = LocaleProvider('./zope/i18n/locales/data')
>>> locale = provider.getLocale('en', 'US')
```

You can now for example retrieve the currency that is used in the US and get the symbol and name of the currency:

```
>>> numbers = locale.numbers
>>> currency = numbers.currencies['USD']
>>> currency.symbol
u'$'
>>> currency.type
u'USD'
>>> currency.displayName
u'US Dollar'
```

The more interesting tasks are formatting and parsing dates/times. There are four predefined date/time formatters that you can choose from: "short", "medium", "full", and "long". Here we just use "short":

```
>>> formatter = locale.dates.getFormatter('dateTime', length='short')
>>> formatter.parse(u'1/25/80 4:07 AM')
datetime.datetime(1980, 1, 25, 4, 7)
>>> from datetime import datetime
>>> dt = datetime(1980, 1, 25, 4, 7, 8)
>>> formatter.format(dt)
u'1/25/80 4:07 AM'
```

For numbers you can choose between "decimal", "percent", "scientific", and "currency":

```
>>> formatter = locale.numbers.getFormatter('decimal')
>>> formatter.parse(u'4,345.03')
4345.0299999999997
>>> formatter.format(34000.45)
u'34,000.45'
```

10.4 Messages and Message Catalogs

While the object formatting is the more interesting task, the more common one is the markup and translation of message strings. In order to manage translations better, message strings are categorized in domains. There is currently only one domain for all of the Zope core called "zope". Products, such as ZWiki, would use a different

domain, such as "zwiki". Translatable messages are particularly marked in the code (see the section below) and are translated before their final output.

All message translations for a particular language of one domain are stored in a message catalog. Therefore we have a message catalog for each language and domain pair. We differentiate between filesystem (global) and ZODB (local) product development. Global message catalogs are standard gettext PO files. The PO files for the "zope" domain are located in `ZOPE3/src/zope/app/locales/⟨REGION⟩/LC_MESSAGES/zope.po`, where REGION can be `de`, `en` or `pt_BR`.

Local message catalogs, on the other hand, are managed via the ZMI through local translation domains. In such a utility you can create new languages, domains and message strings, search through existing translations and make changes, import/export external message catalogs (Gettext PO files), and synchronize this translation domain with another one. Especially the synchronization between translation domain utilities is very powerful, since it allows easy translation upgrades between development and production environments.

Okay, now we know how to manage translatable strings, but how can we tell the system which strings are translatable? Translatable strings can occur in ZPT, DTML, ZCML and Python code. We noticed however, that almost all Python-based translatable strings occur in views, which led us to the conclusion that message strings outside views are usually a sign of bad programming and we have only found a few exceptions (like interface declarations). This leads to a very important rule:

Translations of human readable strings should be done very late in the publication process, preferrably just before the final output.

In the next section we will go into some more detail on how to markup the code in each language.

10.5 Internationalizing Message Strings

10.5.1 Python Code

As mentioned before, Zope is not a simple application, and therefore we cannot translate a text message directly in the Python code (since we do not know the user's locale), but must mark them as translatable strings, which are known as `MessageId`s. Message Ids are created using Message Id factories. The factory takes the domain as argument to the constructor:

```
1 from zope.i18nmessageid import MessageIDFactory
2 _ = MessageIDFactory('demo')
```

Note: The `_` (underscore) is a convention used by gettext to mark text as translatable.

Now you can simply mark up translatable strings using the `_` function:

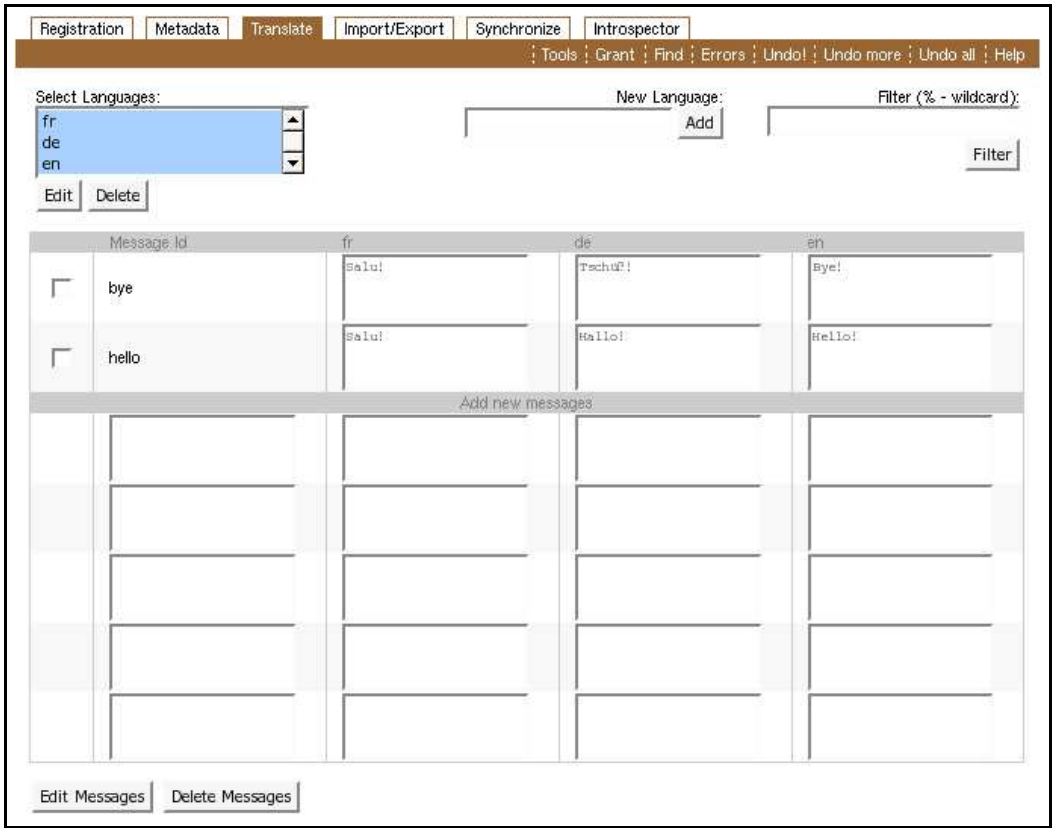


Figure 10.1: This is the main translation screen. It allows you add languages to the domain to edit. Once a language is created, you need to select it for modification. You can then add new messages or edit existing ones of the languages you selected. The filter option will help you to filter the message ids, so that you can find messages faster.

```
1 title = _('This is the title of the object.')
```

But this is the simple case. What if you want to include some data? Then you can use:

```
1 text = _('You have $x items.')
2 text.mapping = {'x': x}
```

In this case the number is inserted after the translation. This way you can avoid having a translation for every different value of `x`.

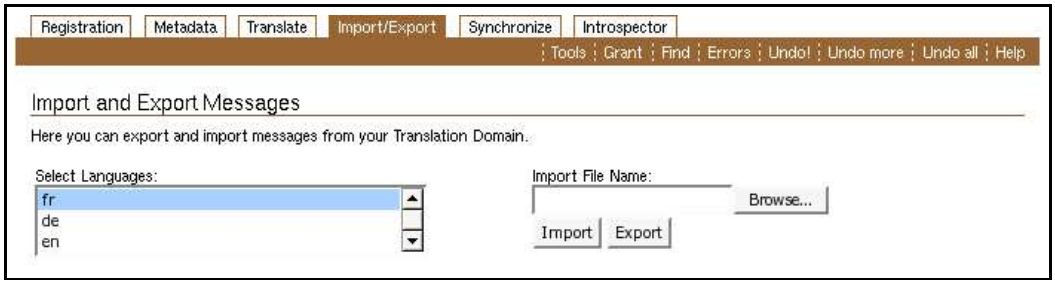


Figure 10.2: Using this screen you can import Gettext PO files or export the existing translations to Gettext PO files.

10.5.2 ZPT (Page Templates)

For Page Templates we developed a special `i18n` namespace (as mentioned before), which can be used to translate messages. The namespace is well documented at `http://dev.zope.org/Zope3/ZPTInternationalizationSupport` and some examples can be found at `http://dev.zope.org/Zope3/ZPTInternationalizationExamples`.

10.5.3 DTML

There is no DTML tag defined for doing translations yet, but we think it will be very similar to the ZBabel and Localizer version, since they are almost the same.

10.5.4 ZCML

I briefly described ZCML's way of internationalizing text in the previous chapter. In the schema of each ZCML directive you can declare translatable attributes simply by making them `MessageId` fields. The domain for the message strings is provided by the `i18n_domain` attribute in the configure tag. Therefore the user only has to specify this attribute to do the I18n in ZCML.

Once the code is marked up, you must extract these strings from the code and compile message catalogs. For this task there is a tool called `ZOPE3/utilities/i18nextract.py`. Its functionality and options are discussed in "Internationalizing a Product".

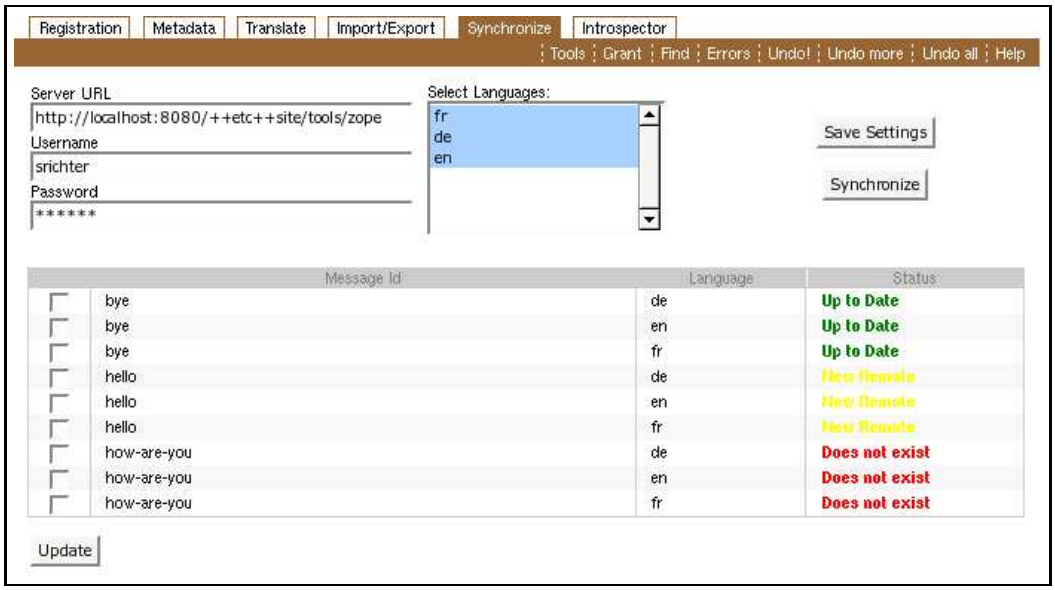


Figure 10.3: With the Synchronization screen, you can synchronize translations accross remote domains.

Chapter 11

Meta Data and the Dublin Core

Difficulty

Newcomer

Skills

- It is necessary that you are familiar with the previous chapters of this part, specifically the introduction to components.
- You should be familiar with the term "meta-data" and what it implies.
- Be knowledgeable about the Dublin Core standard. Optional.

Problem/Task

Any advanced system has the need to specify meta-data for various artifacts of its system, especially for objects that represent content. For a publishing environment like Zope 3 it is important to have a standard set of meta-data fields for all content objects. Already in Zope 2's CMF, the Dublin Core was used to provide such a set of fields.

Solution

Even though I expect that you know what the term "meta-data" means, it can be useful to do a quick review since people use the term in a very broad sense. Data in general is the information an object inheritly carries. It represents the state and is necessary to identify the object. Meta-data on the other hand is information *about* the object and its content. It is not required for the object to function in itself

(i.e. object methods should not depend on meta-data) but the meta-data might be important for an object to function inside a larger framework, providing additional information for identification, cataloging, indexing, integration with other systems, etc.

One standard set of meta-data is called "Dublin Core" (`dublincore.org`). The Dublin Core provides additional information about content-centric objects, such as the title, description (summary or abstract) and author of the object. As said before, Dublin Core was very successful in Zope 2's CMF and Plone.

In the Dublin Core, short DC, all elements are lists, meaning that they can have multiple values. The DC elements are useful to us, because they cover the very most common meta-data items, like the creation date, title, and author. This data is useful in the context of most objects and at least their high-level meaning is easily understood. But there are some issues with the Dublin Core as well. There is a temptation for developers to interpret some deep meaning into the DC elements, since it is such a well-established standard. As Ken Manheimer pointed out, even the Dublin Core designers succumbed to that temptation, and tried to be a bit too ambitious, with some of the fields.

A good example here is the contributor element. It is not clear what is meant by a contributor. Is it an editor, translator, or an additional content author? And how does this information help me, if I want to find the person who last modified the object or publication? Therefore it becomes important to specify the meaning of the various elements (and the items in a particular element) for each specific implementation, such as Zope 3. All the elements and how they are implemented are well documented by the interfaces found in `ZOPE3/src/zope/app/interfaces/dublincore.py` and in the section below.

The Dublin Core Elements

The following Dublin Core element list was taken from `http://dublincore.org/documents/2003/02/04/dces/`. I added and edited some more comments with regard to Zope 3's implementation.

Title

Label

Title

Definition

A human-readable name given to the resource.[1]

Comment

In the Zope 3, the name of a resource is a unique string within its container (it used to be called "id" in Zope 2). However, names of objects are often not presented to the end user. The title is used to represent an object instead.

Creator

Label

Creator

Definition

An entity primarily responsible for making the content of the resource.

Comment

A creator in Zope is a special example of a principal, which can take a lot of forms, but it will typically be a user of the application. Zope 3 stores the user id in this field.

Subject

Label

Subject and Keywords

Definition

A topic of the content of the resource.

Comment

Typically, Subject will be expressed as keywords, key phrases or classification codes that describe a topic of the resource. Recommended best practice is to select a value from a controlled vocabulary or formal classification scheme. Note that this is ideal for cataloging.

[1] Resource is the generic term for an object in the Dublin Core language and has nothing to do with Zope 3's concept of a resource, which is a presentation component that does not require a context. You should read "content component" instead of resource for all occurrences of the term in this chapter.

Description

Label

Description

Definition

An account of the content of the resource.

Comment

Examples of Description include, but is not limited to: an abstract, table of contents, reference to a graphical representation of content or a free-text account of the content. In Zope 3 we usually use the description to give some more details about the semantics of the object/resource, so that the user gains a better understanding about its purpose.

Publisher

Label

Publisher

Definition

An entity responsible for making the resource available

Comment

It is unlikely that this entity will be used heavily in Zope 3, but it might be useful for workflows of News sites and other publishing applications. The Publisher is the name/id of a principal.

Contributor

Label

Contributor

Definition

An entity responsible for making contributions to the content of the resource.

Comment

Examples of Contributor include a person, an organization, or a service. Typically, the name of a Contributor should be used to indicate the entity. As mentioned before, this term is incredibly vague and needs some additional policy; Zope 3 has not made up such a policy yet. The Contributor is the name/id of a principal.

Date

Label

Date

Definition

A date of an event in the lifecycle of the resource.

Comment

Typically, Date will be associated with the creation or availability of the resource. Recommended best practice for encoding the date value is defined in a profile of ISO 8601 [W3CDTF] and includes (among others) dates of the form YYYY-MM-DD. Note, that often time matters to us as well; of course, instead of saving text we store Python datetime objects. Also note that the definition is very vague and needs some more policy to be useful.

Type

Label

Resource Type

Definition

The nature or genre of the content of the resource.

Comment

Type includes terms describing general categories, functions, genres, or aggregation levels for content. Recommended best practice is to select a value from a controlled vocabulary (for example, the DCMI Type Vocabulary [DCT1]). To describe the physical or digital manifestation of the resource, use the "Format" element. For content objects the resource type is clearly the "Content Type". For other objects it

might be simply the registered component name. However, Zope 3 is not using this element yet.

Format

Label

Format

Definition

The physical or digital manifestation of the resource.

Comment

Typically, Format may include the media-type or dimensions of the resource. Format may be used to identify the software, hardware, or other equipment needed to display or operate the resource. Examples of dimensions include size and duration. Recommended best practice is to select a value from a controlled vocabulary (for example, the list of Internet Media Types [MIME] defining computer media formats). We have not used this element so far, even though I think we could use some of the existing framework for this.

Identifier

Label

Resource Identifier

Definition

An unambiguous reference to the resource within a given context.

Comment

Recommended best practice is to identify the resource by means of a string or number conforming to a formal identification system. Formal identification systems include but are not limited to the Uniform Resource Identifier (URI) (including the Uniform Resource Locator (URL)), the Digital Object Identifier (DOI) and the International Standard Book Number (ISBN). In Zope 3's case this could be either the object's path or unique id (as assigned by some utility.

Source

Label

Source

Definition

A Reference to a resource from which the present resource is derived.

Comment

The present resource may be derived from the Source resource in whole or in part. Recommended best practice is to identify the referenced resource by means of a string or number conforming to a formal identification system. I do not see how this is generically useful to Zope components, though I think is could be applicable in specific applications written in Zope.

Language

Label

Language

Definition

A language of the intellectual content of the resource.

Comment

Recommended best practice is to use RFC 3066 [RFC3066] which, in conjunction with ISO639 [ISO639]), defines two- and three primary language tags with optional subtags. Examples include "en" or "eng" for English, "akk" for Akkadian, and "en-GB" for English used in the United Kingdom. Note that we have a system in place to describe locales; see the introduction to internationalization and localization.

Relation

Label

Relation

Definition

A reference to a related resource.

Comment

Recommended best practice is to identify the referenced resource by means of a string or number conforming to a formal identification system. Another vague element, since it does not specify what sort of relation is meant; it could be "containment" for example, in which case the parent object would be a good candidate. However, more policy is required to make the field useful to Zope.

Coverage

Label

Coverage

Definition

The extent or scope of the content of the resource.

Comment

Typically, Coverage will include spatial location (a place name or geographic coordinates), temporal period (a period label, date, or date range) or jurisdiction (such as a named administrative entity). Recommended best practice is to select a value from a controlled vocabulary (for example, the Thesaurus of Geographic Names [TGN]) and to use, where appropriate, named places or time periods in preference to numeric identifiers such as sets of coordinates or date ranges. This seems not to be useful to Zope generically.

Rights

Label

Rights Management

Definition

Information about rights held in and over the resource.

Comment

Typically, Rights will contain a rights management statement for the resource, or reference a service providing such information. Rights information often encompasses Intellectual Property Rights (IPR), Copyright, and various Property Rights. If the

Rights element is absent, no assumptions may be made about any rights held in or over the resource. Zope 3 could use this element to show its security settings on this object, in other words who has read and modification access to the resource. It makes little sense to use this element to generically define a copyright or license entry. Again, specific applications might have a better use for this element.

Chapter 12

Porting Applications from Zope 2 to Zope 3

Difficulty

Sprinter

Skills

- You should have some understandings of Zope 3 and know how to develop filesystem-based products for it. If necessary read part C and D of this book first.
- You should know how to develop Zope 2 products.
- It is usefull to know the the ZWiki for Zope 2 product and its purpose.
- Be familiar with the Zope 2 implementation of ZWiki. Optional.

Problem/Task

Porting applications from an older to a new version of a framework is always tricky, espsecially if new version is a total rewrite and based on a totally different software model. But Zope 3 is not only a complete rewrite of Zope 2, but also a complete shift in philosophy and development style.

There are two methods to approach porting an application. The first one is to completely redesign the existing Zope 2 application in light of the Zope 3 framework. The second method uses various compatibility layers and conversions scripts to allow the existing code to run under Zope 3.

Solution

12.1 Porting an Application by Redesign

The great advantage of redesigning and reimplementing an application is that one can make use of all the new tools and gadgets available under the new framework. In the case of Zope 3, this tends to make the new implementation much cleaner and to consist of less code. The disadvantage here is that this solution will take a lot of time, since usually all the logic must be written from scratch. However, the GUI templates are usually less affected and will be reusable.

This section is based on my experience with porting the core functionality of the ZWiki product for Zope 2 to Zope 3 for which I had some help from Simon Michael, the original author of ZWiki. It turns out that this section also serves well as a comparison between the development models of Zope 2 and 3.

Just as a review: What are Wikis? Wikis are easily editable Web pages that are linked through keywords. If a word in a Wiki page is identical to the name of another Wiki page, then a so-called Wiki link is inserted at this position making a connection to the other Wiki page. If a mixed-case term is found in a Wiki Page (like "Zope3Book"), then a question mark is placed behind it. If you click it you are able to create a new Wiki Page with the name "Zope3Book". Every user of the Wiki can usually update and add new content and Wiki pages. These properties make Wikis a great tool for interactive and networked technical documentation and communication.

In the Zope 2 implementation, most of the ZWiki functionality is in a gigantic 121 kB file called `ZWikiPage.py`, including core and advanced functionality, Zope 2 presentation logic as well as documentation. It was almost impossible for me to understand what's going on and the entire code did not seem very Pythonic. For example, all possible source to ouput converters (STX to HTML, Plain Text to HTML, STX with DTML to HTML, etc.) were hardcoded into this one massive class and new converters could only be added through monkey patching. The code was so massive, that I had to ask Simon to find the core functionality code for me. However, I do not think that the messy code is his fault though; it is a classic example of the break down of the of the Zope 2 development model. Since Zope 2 encourages development of extensions through inheritance, old applications often consist of massive classes that eventually become unmanagable.

But where do we start with the refactoring? The first task is to (re)identify the core functionality of the object you try to port. For the `ZWikiPage` class the main purpose is to store the source of the page and maybe the type of the source, i.e. Plain Text, STX, reST or whatever. Note that the object itself has no knowledge

whatsoever about how to convert from the various source types to HTML for example, since these converters are presentation specific. The new implementation of the `WikiPage` object is now only 26 lines long! All additional, non-core features will be added through adapters and views later.

Okay, we have a nice compact content object, but now we have to add some presentation logic. Since we used schemas to define the attributes of the `WikiPage` object, we can create add and edit screens purely using ZCML directives, which means no further Python or ZPT code. The tricky part to get the converters right. I decided to have an `ISource` interface, which would serve as a base for all possible source types (Plain Text, STX, ReST, etc.) and an `ISourceRenderer`, which would serve as a base to render a particular source type to a particular presentation type (i.e. Browser, XUL, ...). That means that renderers are simply views of `ISource` objects. To ease the implementation of new source renderers for other developers, new ZCML directives were developed to hide the low-level management of source types and renderers. The renderer code was later placed in a seperate package (`zope.app.renderer`), since it is useful for many other applications as well. See the interfaces in `zope.app.renderer.interfaces` for details.

The result was very nice. With some abstraction and new configuration code I was able to make the content object to be totally agnostic of the rendering mechanism. Furthermore, it is easy to add new third party source types and renderers without altering or monkey patching the original product. Add-on features, such as a page hierarchy and E-mail notification support have been implemented using adapters and annotations. See the chapters in the two Content Components parts for more details on how to develop a product from scratch and extending it without touching the original code.

The Zope 3 Wiki implementation is part of the Zope 3 core and can be found at `ZOPE3/src/zwiki`.

12.2 Porting using compatibility layers and scripts

The advantage of using conversion scripts and compatibility layers is that porting an application should be fast and easy. However, while this saves a lot of initial development time, such a solution will not facilitate the usage of the new features of the framework and one might keep a lot of unnecessary code around.

There are currently no compatibility layers and scripts available for porting Zope 2 applications to Zope 3. The current plan is that they will be written after Zope X3.0 is out of the door.

A final alternative for porting applications is to use Zope 2 and 3 at the same time and convert features slowly as you can. The “Five” project, initiated by Martijn

Faassen, in a Zope 2 product that makes Zope 3 available under Zope 2 and allows one to use ZCML to configure Zope 2. See `http://codespeak.net/z3/five.html` for the project's homepage.

PART III

Content Components – The Basics

This section deals with the creation of content objects and basic functionality around them. In order to make the chapters flow better, they will be all guided by the creation of a simple message board application.

Chapter 13: Writing a new Content Object

This chapter will describe how to implement a new simple content component/object.

Chapter 14: Adding Views

This chapter will demonstrate various methods to create Browser-specific views for a component.

Chapter 15: Custom Schema Fields and Form Widgets

This chapter basically tells you how to implement your own field and corresponding widget. It will then be demonstrated how this field and widget can be used in a content object.

Chapter 16: Securing Components

Zope 3 comes with an incredible security system; but the best system is only as good as the end developer using it. This chapter will give some hands-on tips and tricks on how to make your code secure.

Chapter 17: Changing Size Information

There exists a small interface for content objects that will allow them to display and compare their size. This is a short chapter explaining this feature.

Chapter 18: Internationalizing a Package

This chapter will give step by step instructions on how to internationalize the application we developed in the previous chapters and how to create a German translation for it.

CHAPTER 13

WRITING A NEW CONTENT OBJECT

Difficulty

Newcomer

Skills

- The developer should be familiar with Python and some object-oriented concepts. Component-based programming concepts are a plus.

- Some understanding of the schema and interface packages. Optional.

Problem/Task

Of course it is essential for any serious Zope 3 developer to know how to implement new content objects. Using the example of a message board, this chapter will outline the main steps required to implement and register a new content component in Zope 3.

Solution

This chapter is the beginning of the development of `MessageBoard` content type and everything that is to it. It serves very well as the first hands-on task, since it will not assume anything other than that you have Zope 3 installed, know some Python and are willing to invest some time in learning the framework.

13.1 Step I: Preparation

Before you start, you should have installed Zope 3, created a `principal.zcml` file and have successfully started Zope. You have already done that? Okay, then let's start.

Other than in Zope 2, Zope 3 does not require you to place add-on packages in a special directory and you are free to choose where to put it. The most convenient place is inside `ZOPE3/src` (`ZOPE3` is your Zope 3 directory root), since it does not require us to mess with the `PYTHONPATH`. To clearly signalize that this application is demo from the book, we place all of the code in a package called `book`. To create that package, add a directory using

```
mkdir ZOPE3/src/book
```

on Unix.

To make this directory a package, place an empty `__init__.py` file in the new directory. In Unix you can do something like

```
echo "# Make it a Python package" >> ZOPE3/src/book/__init__.py
```

but you can of course also just use a text editor and save a file of this name. Just make sure that there is valid Python code in the file. The file should at least contain some whitespace, since empty files confuse some archive programs.

Now we create another package inside `book` called `messageboard`, in a similar manner (do not forget to create the `__init__.py` file). From now on we are only going to work inside this `messageboard` package, which should be located at `ZOPE3/src/book/messageboard`.

Note: While the source code, that you can download for every step at `http://svn.zope.org/book/trunk/messageboard`, contains a license header, we omit these throughout the book to save typing and space. However, the copyright as stated in the source files still applies.

13.2 Step II: The Initial Design

As stated above, our goal is to develop a fully functional, though not great-looking, Web-based message board application. The root object will be the `MessageBoard`, which can contain postings or `Message` objects from various users. Since we want to allow people to respond to various messages, we need to allow messages to contain replies, which are in turn just other `Message` objects.

That means we have two container-based components: The `MessageBoard` contains only messages and can be added to any Folder or container that wishes to be

able to contain it. To make the message board more interesting, it also has a description, which briefly introduces the subject/theme of the discussions hosted. Messages, on the other hand should be only contained by message boards and other messages. They will each have a title and a body.

This setup should contain all the essential things that we need to make the object usable. Later on we will associate a lot of other meta-data with these components to integrate them even better into Zope 3 and add additional functionality.

13.3 Step III: Writing the interfaces

The very first step of the coding process is always to define your interfaces, which represent your external API. You should be aware that software that is built on top of your packages expect the interfaces to behave exactly the way you specify them. This is often less of an issue for attributes and arguments of a method, but often enough developers forget to specify what the expected return value of a method or function is or which exceptions it can raise or catch.

Interfaces are commonly stored in an `interfaces` module or package. Since our package is not that big, we are going to use a file-based module; therefore start editing a file called `interfaces.py` in your favorite editor.

In this initial step of our application, we are only interested in defining one interface for the message board itself and one for a single message, which are listed below (add these to the file `interfaces.py`):

```
1  from zope.interface import Interface
2  from zope.schema import Text, TextLine, Field
3
4  from zope.app.container.constraints import ContainerTypesConstraint
5  from zope.app.container.constraints import ItemTypePrecondition
6  from zope.app.container.interfaces import IContained, IContainer
7  from zope.app.file.interfaces import IFile
8
9
10 class IMessage(Interface):
11     """A message object. It can contain its own responses."""
12
13     def __setitem__(name, object):
14         """Add a IMessage object."""
15
16     title = TextLine(
17         title=u"Title/Subject",
18         description=u"Title and/or subject of the message.",
19         default=u"",
20         required=True)
21
22     body = Text(
23         title=u"Message Body",
24         description=u"This is the actual message. Type whatever you wish.",
25         default=u"",
```

```
26          required=False)
27
28
29  class IMessageBoard(IContainer):
30      """The message board is the base object for our package. It can only
31      contain IMessage objects."""
32
33      def __setitem__(name, object):
34          """Add a IMessage object."""
35
36      __setitem__.precondition = ItemTypePrecondition(IMessage)
37
38      description = Text(
39          title=u"Description",
40          description=u"A detailed description of the content of the board.",
41          default=u"",
42          required=False)
43
44
45  class IMessageContained(IContained):
46      """Interface that specifies the type of objects that can contain
47      messages."""
48      __parent__ = Field(
49          constraint = ContainerTypesConstraint(IMessageBoard, IMessage))
50
51
52  class IMessageContainer(IContainer):
53      """We also want to make the message object a container that can contain
54      responses (other messages) and attachments (files and images)."""
55
56      def __setitem__(name, object):
57          """Add a IMessage object."""
58
59      __setitem__.precondition = ItemTypePrecondition(IMessage, IFile)
```

▷ Line 1: Import the base Interface class. Any object that has this meta-class in its inheritance path is an interface and *not* a regular class.

▷ Line 2: The attributes and properties of an object are described by fields. Fields hold the meta-data about an attribute and are used, among other things, to validate values and create auto-generated input forms. Most fields are defined in the `zope.schema` package. For more details and a complete list of fields see "Zope Schemas and Widgets (Forms)".

▷ Line 4: `ContainerTypesConstraint` conditions allow us to tell the system to which type of containers an object can be added. For example, a message only wants to be contained by the message board and another message (when it is a reply to the parent message). See below how it is used.

▷ Line 5: The `ItemTypePrecondition` is the opposite of the container types constraint in that it specifies the object types that can be contained by a container. See below for its usage.

- Line 6: Objects providing the `IContained` interface can be included in the Zope object tree. We also import `IContainer` here, which is used as a base interface in Line 29 and 52. `IContainer` defines all necessary methods for this object to be recognized as a container by Zope 3.

  Note that we do not need to inherit `Interface` directly, since `IContainer` already inherits it, which automatically makes `IMessageBoard` also an interface.

- Line 10: You might have already noticed the "I" in front of all the interfaces, which simply stands for "Interface" as you might have guessed already. It is a convention in Zope, so that we do not confuse interfaces and classes, since these two different object types cannot be used in the same way.

  In general messages simply are objects that have a title and a body. Nothing more. We later declare more semantics through additional interfaces and meta-data.

- Line 16–20: A simple title/subject headline for the message. Note that we made this a `TextLine` instead of a `Text` field, so that no newline characters can be inserted. This way the title will be relatively short and will be perfect for the title display where desired.

- Line 22–26: The body is the actual message content. Note that we made no restriction to its size, which you might want to do, in case you are afraid of spam filling your message board.

- Line 33–36: We do not want to allow any content type to be added to a message board. In fact, we just want to be able to add `IMessage` objects. Therefore we declare a precondition on the `__setitem__()` method of the message board interface. Simply list all allowed interfaces as arguments of the `ItemTypePrecondition` constructor.

  Note: Even though `IContainer` already defined `__setitem__()`, we have to declare it here again, so that it is in the scope of the interface and specific to the `IMessageBoard`; otherwise all `IContainer` objects will have this precondition.

- Line 38–42: Declare a property on the interface that will contain the description of the message board. It is a typical `Text` field with the usual options (see "Zope Schemas and Widgets (Forms)" for details). One note though: Notice that we *always* use unicode strings for human-readable text – this is a required convention throughout Zope 3. One of the major focus points of Zope 3 is internationalization and unicode strings are the first requirement to support multi-lingual applications.

- Line 45–49: This interface describes via field constraint which other content types can contain a message. Clearly message boards can contain messages, but also messages can contain other messages – known as responses. We usually specify

this constraint on the parent on the main content type interface (i.e. `IMessage`) directly, but since this constraint refers explicitely to `IMessage` we have to wait till the interface is defined.

▷ Line 52–59: We also want the message to be container, so it can contain responses and attachments. However, we do not want any object to be able to be added to messages, so we add a precondition as we did for the `IMessageBoard` interface. Again, we have to do this in a separate interface here, since we reference `IMessage` in the condition.

13.4 Step IV: Writing Unit tests

There are two ways unit tests can be written in Zope 3. The first one is through special `TestCase` classes using the `unittest` package, which was modeled after JUnit. The second technique is writing tests inside doc strings, which are commonly known as doc tests.

Late in the development of Zope 3 doc tests became the standard way of writing tests. For philosophical and technical differences between the two approaches, see the section on "Writing Tests", especially the "Writing Basic Unit Tests" and "Doctests: Example-driven Unit Tests" chapters.

Common unit tests, however, are of advantage when it is desirable to reuse abstract tests, as it is the case for various container tests. Therefore, we will use unit tests for the container tests and doc tests for everything else.

First, create a package called `tests` inside the `messageboard` package. Note that calling the test module `tests` (file-based test modules would be called `tests.py`) is a convention throughout Zope 3 and will allow the automated test runner to pick up the tests.

Next, start to edit a file called `test_messageboard.py` and insert:

```
1  import unittest
2  from zope.testing.doctestunit import DocTestSuite
3
4  from zope.app.container.tests.test_icontainer import TestSampleContainer
5
6  from book.messageboard.messageboard import MessageBoard
7
8
9  class Test(TestSampleContainer):
10
11     def makeTestObject(self):
12         return MessageBoard()
13
14 def test_suite():
15     return unittest.TestSuite((
16         DocTestSuite('book.messageboard.messageboard'),
17         unittest.makeSuite(Test),
```

```
18         ))
19
20 if __name__ == '__main__':
21     unittest.main(defaultTest='test_suite')
```

A lot of cool stuff just happened here. You just got your first 12 unit tests. Let's have a closer look:

- Line 1: The `unittest` module comes with stock Python and is used to create the test suite.
- Line 2: Zope provides a specialized `DocTestSuite` that integrates doc tests into the common `unittest` framework and allows the doc tests to be run via the test runner.
- Line 4: There are some basic tests for containers, so we should import them. Freebie tests are always good.
- Line 9–13: Define the Container tests. We only have to provide an instance of the container we would like to be tested as the return value of the `makeTestObject()` method.
- Line 15–19: The `test_suite()` method collects all the defined test cases and compiles them into one test suite. This method must always be named that way, so that the test runner picks up the suite.

  Besides the container test, we also already register the doc tests.
- Line 21–23: We also want to allow any test module to be executable by itself. Here we just tell the test runner to execute all tests of the test suite returned by `test_suite()`. These lines are common boilerplate for any test module in Zope 3.

Now it is time to do the second test module for the `IMessage` component. To start, we simply copied the `test_messageboard.py` to `test_message.py` and modified the new file to become:

```
1  import unittest
2  from zope.testing.doctestunit import DocTestSuite
3
4  from zope.app.container.tests.test_icontainer import TestSampleContainer
5
6  from book.messageboard.message import Message
7
8
9  class Test(TestSampleContainer):
10
11     def makeTestObject(self):
12         return Message()
13
14 def test_suite():
```

```
15      return unittest.TestSuite((
16          DocTestSuite('book.messageboard.message'),
17          unittest.makeSuite(Test),
18          ))
19
20  if __name__ == '__main__':
21      unittest.main(defaultTest='test_suite')
```

There is not really any difference between the two testing modules, so that I am not going to point out the same facts again.

Note that none of the tests deal with implementation details yet, simply because we do not know what the implementation details will be. These test could be used by other packages, just as we used the `SampleContainer` base tests, since these tests only depend on the API. In general, however, tests should cover implementation-specific behavior.

13.5 Step V: Implementing Content Components

Now we are finally ready to implement the content components of the package. This is the heart of this chapter. But how do we know which methods and properties we have to implement? There is a neat tool called `pyskel.py` in `ZOPE3/utiltities` that generates a skeleton. Go to `ZOPE3/src` and type:

```
python2.3 ../utilities/pyskel.py \
          book.messageboard.interfaces.IMessageBoard
```

The expected result is shown below. The tool inspects the given interface and creates the skeleton of an implementing class. It also recurses into all base interfaces to get their methods. Here the generated code:

```
1  from zope.interface import implements
2  from book.messageboard.interfaces import IMessageBoard
3
4  class MessageBoard:
5      __doc__ = IMessageBoard.__doc__
6
7      implements(IMessageBoard)
8
9
10     def __setitem__(self, name, object):
11         "See book.messageboard.interfaces.IMessageBoard"
12
13     # See book.messageboard.interfaces.IMessageBoard
14     description = None
15
16     def __getitem__(self, key):
17         "See zope.interface.common.mapping.IItemMapping"
18
19     def get(self, key, default=None):
20         "See zope.interface.common.mapping.IReadMapping"
21
```

```
22      def __contains__(self, key):
23          "See zope.interface.common.mapping.IReadMapping"
24
25      def __getitem__(self, key):
26          "See zope.interface.common.mapping.IItemMapping"
27
28      def keys(self):
29          "See zope.interface.common.mapping.IEnumerableMapping"
30
31      def __iter__(self):
32          "See zope.interface.common.mapping.IEnumerableMapping"
33
34      def values(self):
35          "See zope.interface.common.mapping.IEnumerableMapping"
36
37      def items(self):
38          "See zope.interface.common.mapping.IEnumerableMapping"
39
40      def __len__(self):
41          "See zope.interface.common.mapping.IEnumerableMapping"
42
43      def get(self, key, default=None):
44          "See zope.interface.common.mapping.IReadMapping"
45
46      def __contains__(self, key):
47          "See zope.interface.common.mapping.IReadMapping"
48
49      def __getitem__(self, key):
50          "See zope.interface.common.mapping.IItemMapping"
51
52      def __setitem__(self, name, object):
53          "See zope.app.container.interfaces.IWriteContainer"
54
55      def __delitem__(self, name):
56          "See zope.app.container.interfaces.IWriteContainer"
```

This result is good but some parts are unnecessary; we will for example simply inherit the `BTreeContainer` base component, so that we do not have to implement the methods from the `IReadMapping`, `IEnumerableMapping`, `IReadMapping`, `IItemMapping` and `IWriteContainer` interfaces.

Open a new file called `messageboard.py` for editing. The implementation of the message board including doc tests looks like this:

```
1  from zope.interface import implements
2  from zope.app.container.btree import BTreeContainer
3
4  from book.messageboard.interfaces import IMessageBoard
5
6  class MessageBoard(BTreeContainer):
7      """A very simple implementation of a message board using B-Tree Containers
8
9      Make sure that the ``MessageBoard`` implements the ``IMessageBoard``
10     interface:
11
12     >>> from zope.interface.verify import verifyClass
13     >>> verifyClass(IMessageBoard, MessageBoard)
```

```
14      True
15
16      Here is an example of changing the description of the board:
17
18      >>> board = MessageBoard()
19      >>> board.description
20      u''
21      >>> board.description = u'Message Board Description'
22      >>> board.description
23      u'Message Board Description'
24      """
25      implements(IMessageBoard)
26
27      # See book.messageboard.interfaces.IMessageBoard
28      description = u''
```

▷ Line 1: The `implements()` method is used to declare that a class implements one or more interfaces. See "An Introduction to Interfaces" for more details.

▷ Line 2: Everything that has to do with containers is located in `zope.app.container`. `BTreeContainer`s are a very efficient implementation of the `IContainer` interface and are commonly used as base classes for other container-ish objects, such as the message board.

▷ Line 7–24: The class docstring's purpose is to document the class. To follow Python documentation standards, all docstrings should be using the re-structured text format. And doc tests are considered documentation, so it should be written in a narrative style.

On line 12–14 we verify that the `MessageBoard` component really implements `IMessageBoard`. The `verifyClass` function actually checks the object for the existence of the specified attributes and methods.

Lines 18 through 23 just give a demonstration about the default `description` value and how it can be set. The test seems trivial, but at some point you might change the implementation of the `description` attribute to using properties and the test should still pass.

▷ Line 25: Here we tell the class that it implements `IMessage`. This function call might seem like magic, since one might wonder how the function knows as to which class to assign the interface. For the ones interested, it uses `sys.getframe()`.

▷ Line 27–28: Make the description a simple attribute.

Note: Python is very unique in this way. In almost any other object-oriented language (for example Java) one would have written an accessor (`getDescription()`) and a mutator (`setDescription(desc)`) method. However, Python's attribute and property support makes this unnecessary, which in turn makes the code cleaner.

The next task is to write the `Message` object, which is pretty much the same code. Therefore we will not list it here and refer you to the code at `http://svn.zope.org/book/trunk/messageboard/step01/message.py`. The only difference is that in this case the `Message` component must implement `IMessage`, `IMessageContained`, and `IMessageContainer`.

13.6 Step VI: Running Unit Tests against Implementation

After having finished the implementation, we need to make sure that all the tests pass. There is a script called `test.py` that will run all or only specified tests for you. To run the test against your implementation, execute the following line from the Zope 3 root directory:

```
python2.3 test.py -vpu --dir src/book/messageboard
```

The `-v` option cases the currently running test to be displayed, the `-p` allows us to see the progress of the tests being run and `-u` tells the test runner to just run the unit tests. For a list of all available options run the script with the `-h` (help) option.

You should see 26 tests pass. The output at the of the test run should look like this:

```
Configuration file found.
Running UNIT tests at level 1
Running UNIT tests from /opt/zope/Zope3/Zope3-Cookbook
  26/26 (100.0%): test_values (....messageboard.tests.test_messageboard.Test)
----------------------------------------------------------------------
Ran 26 tests in 0.346s

OK
```

It is very likely that some tests are failing or the test suite does not even run due to syntax errors. This is totally normal and exactly the reason we write tests in the first place. In these cases keep fixing the problems until all tests are passing.

13.7 Step VII: Registering the Content Components

Now that we have developed our components, it is necessary to tell Zope 3 how to interact with them. This is commonly done using Zope's own configuration language called ZCML. The configuration is stored in a file called `configure.zcml` by convention. Start to edit this file and add the following ZCML code:

```
1 <configure
2     xmlns="http://namespaces.zope.org/zope">
3
4   <interface
```

```
5       interface=".interfaces.IMessageBoard"
6       type="zope.app.content.interfaces.IContentType"
7       />
8
9   <content class=".messageboard.MessageBoard">
10    <implements
11        interface="zope.app.annotation.interfaces.IAttributeAnnotatable"
12        />
13    <implements
14        interface="zope.app.container.interfaces.IContentContainer"
15        />
16    <factory
17        id="book.messageboard.MessageBoard"
18        description="Message Board"
19        />
20    <require
21        permission="zope.ManageContent"
22        interface=".interfaces.IMessageBoard"
23        />
24    <require
25        permission="zope.ManageContent"
26        set_schema=".interfaces.IMessageBoard"
27        />
28  </content>
29
30  <interface
31      interface=".interfaces.IMessage"
32      type="zope.app.content.interfaces.IContentType"
33      />
34
35  <content class=".message.Message">
36    <implements
37        interface="zope.app.annotation.interfaces.IAttributeAnnotatable"
38        />
39    <implements
40        interface="zope.app.container.interfaces.IContentContainer"
41        />
42    <require
43        permission="zope.ManageContent"
44        interface=".interfaces.IMessage"
45        />
46    <require
47        permission="zope.ManageContent"
48        interface=".interfaces.IMessageContainer"
49        />
50    <require
51        permission="zope.ManageContent"
52        set_schema=".interfaces.IMessage"
53        />
54  </content>
55
56 </configure>
```

▷ Line 1–2, 65: As the file extension promises, configuration is done using XML. All configuration in a ZCML file must be surrounded by the `configure` element. At the beginning of the `configure` element, we list all the ZCML namespaces that

we are going use and define the default one. In this case we only need the generic `zope` namespace. You will get to know many more namespaces as we develop new functionality in the following chapters.

▷ Line 4–7: It is sometimes necessary to categorize interfaces. One type of category is to specify which interface provides a content type for Zope 3. The `zope:interface` directive is used to assign these types on interfaces. Another way to think about it is that interfaces are just components, and components can provide other interfaces.

▷ Line 9–28: The `zope:content` directive registers the `MessageBoard` class as a content component. The element always has only one attribute, `class`, that points to the component's class using a dotted Python path.

- Line 10–12: In order for the object to have a creation and modification date as well as other meta-data (for example the Dublin Core), we need to tell the system that this object can have annotations associated with itself. This is not necessarily required, but is a good habit. See the chapter on "Using Annotations to Store Meta-Data" for details.

  Annotations store add-on data which is also commonly known as meta-data, since it is data that is not necessary for the correct functioning of the object itself. However, meta-data allows an object to be better integrated in the system. Annotations are heavily used in Zope 3.

  In general, the `zope:implements` sub-directive allows you to assert new implemented interfaces on a class. It is totally equivalent to `classImplements(Class, ISomeInterface)` in Python. So why would we want to declare interfaces in ZCML instead of Python? For one, it clutters the Python code and distracts from the actual functionality of the component. Also, when dealing with 3rd party Python packages, we do not want to touch this code, but still be able to make assertions about objects, so that they can be used inside Zope 3 without modification.

  Note that usually only "marker interfaces", interfaces that have no methods and/or properties, are declared via ZCML, since no additional Python code for the implementation of the interface is required.

- Line 13–15: The `IContentContainer` interface is another example of a marker interface. All that it declares is that this container contains ordinary content in content space, which is clearly the case for our message board.

- Line 16–19: The `zope:factory` sub-directive allows us to register a factory named `book.messageboard.MessageBoard` for the `MessageBoard` component.

Every factory needs an `id` (first directive argument) through which the factory can be accessed and executed. However, you are not required to specify an `id`; if you don't, the literal string value of the `zope:content`'s `class` attribute is used, which would be `.messageboard.MessageBoard` in this case.

The `zope:factory` directive also supports two human-readable information strings, `title` and `description`, that can be used for user interfaces.

- Line 20–27: In Zope 3 we differentiate between trusted and untrusted environments. Trusted environments have no security or can easily circumvent security. And example is file-based Python code, which is always trusted. The opposite is true for untrusted environments; here security should apply everywhere and should not be avoidable. All Web and FTP transactions are considered untrusted.

  Of course, we want to use our message board via the Web, since it is the default user interface of Zope 3. To make it usable, we have to declare the minimal security assertions for the properties and methods of our component. Security assertions are done using the `zope:require` and `zope:allow` directive.

  The `require` directive usually starts out with specifying a `permission`. Then we have to decide what we want to protect with this declaration. Here are your choices:

  1. The `attributes` attribute allows us to specify attributes and methods (note that methods are just callable attributes) that can be *accessed* if the user has the specified permission.
  2. `set_attributes` allows you to specify individual attributes that can be *modified* or mutated. Note that you should not list any methods here, since otherwise someone could override a method inserting malicious code.
  3. If you specify one or more interfaces using the `interface` attribute, the directive will automatically extract all declared methods and properties of the interfaces and grant *access* rights to them.
  4. When you specify a set of schemas using the `set_schema` attribute, then all the defined properties in it are granted *modification* rights. Methods listed in the schema will be ignored.

  Note: In ZCML tokens of a list are separated by simple whitespace and not by comma, as you might have expected.

  A somewhat different option to the above choices is the `like_class` attribute, which must be specified without any permission. If used, it simply transfers all the security assertions from the specified class to the class specified in the `zope:content` directive that encloses the security assertions. In our case this is our `MessageBoard` component. The usage of the directive looks like this:

```
1     <require like_class=".message.Message" />
```

Here the `MessageBoard` would simply "inherit" the security assertions made for the `Message` component.

The second security directive, `zope:allow`, either takes a set of `attributes` or `interfaces`. All attributes specified will be publicly available for everyone to access. This is equivalent to requiring someone to have the `zope.Public` permission, which every principal accessing the system automatically possesses.

So now it is easy to decipher the meaning of our two security assertions. We basically gave read and write access to the `IMessageBoard` interface (which includes all `IContainer` methods and the `description` attribute), if the user has the `zope.ManageContent` permission.

▷ Line 30–54: This is the same as above for the `Message` content component.

13.8 Step VIII: Configure some Basic Views

Even though the content components are registered now, nothing interesting will happen, because there exists only a programmatic way of adding and editing the new components. Thus we are going to define some very basic browser views to make the content components accessible via the browser-based user interface.

First create a package called `browser` (do not forget the `__init__.py` file) inside the `messageboard` package. Add a new configuration file, `configure.zcml`, inside `browser` and insert the following content:

```
1  <configure
2      xmlns="http://namespaces.zope.org/browser">
3
4    <addform
5        label="Add Message Board"
6        name="AddMessageBoard.html"
7        schema="book.messageboard.interfaces.IMessageBoard"
8        content_factory="book.messageboard.messageboard.MessageBoard"
9        fields="description"
10       permission="zope.ManageContent"
11       />
12
13   <addMenuItem
14       class="book.messageboard.messageboard.MessageBoard"
15       title="Message Board"
16       description="A Message Board"
17       permission="zope.ManageContent"
18       view="AddMessageBoard.html"
19       />
20
21   <editform
```

```
22      schema="book.messageboard.interfaces.IMessageBoard"
23      for="book.messageboard.interfaces.IMessageBoard"
24      label="Change Message Board"
25      name="edit.html"
26      permission="zope.ManageContent"
27      menu="zmi_views" title="Edit"
28      />
29
30  <containerViews
31      for="book.messageboard.interfaces.IMessageBoard"
32      index="zope.View"
33      contents="zope.View"
34      add="zope.ManageContent"
35      />
36
37  <addform
38      label="Add Message"
39      name="AddMessage.html"
40      schema="book.messageboard.interfaces.IMessage"
41      content_factory="book.messageboard.message.Message"
42      fields="title body"
43      permission="zope.ManageContent"
44      />
45
46  <addMenuItem
47      class="book.messageboard.message.Message"
48      title="Message"
49      description="A Message"
50      permission="zope.ManageContent"
51      view="AddMessage.html"
52      />
53
54  <editform
55      schema="book.messageboard.interfaces.IMessage"
56      for="book.messageboard.interfaces.IMessage"
57      label="Change Message"
58      fields="title body"
59      name="edit.html"
60      permission="zope.ManageContent"
61      menu="zmi_views" title="Edit"
62      />
63
64  <containerViews
65      for="book.messageboard.interfaces.IMessage"
66      index="zope.View"
67      contents="zope.View"
68      add="zope.ManageContent"
69      />
70
71 </configure>
```

▷ Line 2: In this configuration file we do not use the `zope`, but the `browser` namespace, since we want to configure browser-specific functionality. Also note that `browser` is the default namespace, so that our directives do not need the namespace prefix.

Namespaces for ZCML start commonly with `http://namespaces.zope.org/` followed by the short name of the namespace, which is commonly used in this book to refer to namespaces.

▷ Line 4-11: Register an auto-generated "Add" form for the Message Board.

- Line 5: The label is a small text that is shown on top of the screen.
- Line 6: The name of the view. The name is the string that is actually part of the URL.
- Line 7: This defines the schema that will be used to generate the form. The fields of the schema will be used to provide all the necessary meta data to create meaningful form elements.
- Line 8: The content factory is the class/factory used to create the new content component.
- Line 9: The fields are a list of field names that are displayed in the form. This allows you to create forms for a subset of fields in the schema and to change the order of the fields in the form.
- Line 10: Specifies the permission required to be able to create and add the new content component.

▷ Line 13–19: After creating a view for adding the message board, we now have to register it with the add menu, which is done with the `browser:addMenuItem` directive. The `title` is used to display the item in the menu. The important attribute is the `view`, which must match the name of the add form.

▷ Line 21–28: Declaring an edit form is very similar to defining an add form and several of the options/attributes are the same. The main difference is that we do not need to specify a content factory, since the content component already exists.

The `for` attribute specifies the interface for which type of component the edit form is for. All view directives (except the `browser:addform`) require the `for` attribute. If you would like to register a view for a specific implementation, you can also specify the class in the `for` attribute.

We also commonly specify the menu for edit views directly in the directive using the `menu` and `title` attribute as seen on line 27. The `zmi_views` menu is the menu that creates the tabs on the default Web UI. It contains all views that are specific to the object.

▷ Line 30–35: The message board is a container and a quick way to register all necessary container-specific views is to use the `browser:containerViews` directive. Note though that this directive is not very flexible and you should later replace it by writing regular views.

▷ Line 37–69: These are exactly the same directives over again, this time just for the `IMessage` interface.

In order for the system to know about the view configuration, we need to reference the configuration file in `messageboard/configure.zcml`. To include the view configuration, add the following line:

```
1 <include package=".browser" />
```

13.9 Step IX: Registering the Message Board with Zope

At this stage we have a complete package. However, other than in Zope 2, you have to register a new package explicitly. That means you have to hook up the components to Zope 3. This is done by creating a new file in `ZOPE3/package-includes` called `messageboard-configure.zcml`. The name of the file is not arbitrary and must be of the form `*-configure.zcml`. The file should just contain one directive:

```
1 <include package="book.messageboard" />
```

When Zope 3 boots, it will walk through each file of this directory and execute the ZCML directives inside each file. Usually the files just point to the configuration of a package.

13.10 Step X: Testing the Content Component

Congratulations! You have just finished your first little Zope 3 application, which is quiet a bit more than what would minimally be required as you will see in a moment. It is time now to harvest the fruits of your hard work. Start your Zope 3 server now, best using `makerun` from the Zope 3 root. If you get complains about the Python version being used, edit the `Makefile` and enter the correct path to Python's executable. Other errors that might occur are due to typos or misconfigurations. The ZCML interpreter will give you the line and column number of the failing directive in Emacs-friendly format. Try to start Zope 3 again and again until you have fixed all the errors and Zope 3 starts up ending with this output:

```
------
2003-12-12T23:14:58 INFO PublisherHTTPServer zope.server.http (HTTP) started.
        Hostname: localhost
        Port: 8080
------
2003-12-12T23:14:58 INFO PublisherFTPServer zope.server.ftp started.
        Hostname: localhost
        Port: 8021
------
2003-12-12T23:14:58 INFO root Startup time: 11.259 sec real, 5.150 sec CPU
```

Note that you also get some internationalization warnings, which you can safely ignore for now.

Once the server is up and running, go to your favorite browser and display the following URL:
`http://localhost:8080/@@contents.html`

At this point an authentication box should pop up and ask you for your username and password – users are listed in the `principals.zcml`. If you have not added any special user, use "gandalf" as the login name and "123" as password. After the authentication is complete you should be taken to the Zope 3 Web user interface. Under `Add:` you can now see a new entry "Message Board".

Feel free to add and edit a message board object.

Once you created a message board, you can click on it and enter it. You will now notice that you are only allowed to add "Message" objects here. The choice is limited due to the conditions we specified in the interfaces. The default view will be the "Edit" form that allows you to change the description of the board. The second view is the "Contents" with which you can manage the messages of the message board.

Add a "Message" now. Once you added a message, it will appear in the "Contents" view. You can now click on the message. This will allow you to modify the data about the message and add new messages (replies) to it. With the code we wrote so far, you are now able to create a complete message board tree and access it via the Web UI.

Note that you still might get errors, in which case you need to fix them. Most often you have security problems, which narrows the range of possible issues tremendously. Unfortunately, `NotFoundError` is usually converted to `ForbiddenAttributeError`, so be careful, if you see this problem.

Another common trap is that standard error screens do not show the traceback. However, for these situations the `Debug` skin comes in handy – instead of `http://localhost:8080/@@contents.html` use `http://localhost:8080/++skin++Debug/@@contents.html` and the traceback will be shown.

Note: If you make data-structural changes in your package, it might become necessary to delete old instances of the objects/components. Sometimes even this is not enough, so that you have to either delete the parent `Folder` or best delete the `Data.fs` (ZODB) file. There are ways to upgrade gracefully to new versions of objects, but during development the listed methods are simpler and faster.

The code is available in the Zope SVN under `http://svn.zope.org/book/trunk/messageboard/step01`.

CHAPTER 14

ADDING VIEWS

Difficulty

Newcomer

Skills

- Knowledge gained in the "Writing a new Content Object" chapter.
- Some understanding of the presentation components. Optional.

Problem/Task

Now that we have two fully-functional content objects, we have to make the functionality available to the user, since there are currently only three very simple views: add, edit and contents. In this chapter we will create a nice message details screen as well as a threaded sub-branch view for both messages and the message board.

Solution

This chapter revolves completely around browser-based view components for the `MessageBoard` and `Message` classes. Views, which will be mainly discussed here, are secondary adapters. They adapt IRequest and some context object to some output interface (often just `zope.interface.Interface`).

There are several ways to write a view. Some of the dominant ones include:

1. We already learned about using the `browser:addform`, `browser:editform` and `browser:containerViews` directive. These directives are high-level directives

and hide a lot of the details about creating and registering appropriate view components.

Forms can be easily configured via ZCML, as you have done in the previous chapter. Forms are incredibly flexible and allow you any degree of customization.

2. There is a `browser:page` and a `browser:pages` directive, which are the most common directives for creating browser views and groups of views easily. We will use these two directives for our new views.

3. The `zope:view` directive is very low-level and provides functionality for registering multi-views, which the other directives are not capable of doing. However, for the average application developer the need to use this directive might never arise.

14.1 Step I: Message Details View

Let's now start with the creation of the two new browser views, which is the goal of this chapter. While we are able to edit a message already, we currently have no view for simply viewing the message, which is important, since not many people will have access to the edit screen.

The view displaying the details of a message should contain the following data of the message: the title, the author, the creation date/time, the parent title (with a link to the message), and the body.

Writing a view usually consists of writing a page template, some supporting Python view class and some ZCML to insert the view into the system. We are going to start by creating the page template.

14.1.1 (a) Create Page Template

Create a file called `details.pt` in the `browser` package of `messageboard` and fill it with the following content:

```
<html metal:use-macro="views/standard_macros/view">
  <body>
    <div metal:fill-slot="body">

      <h1>Message Details</h1>

        <div class="row">
            <div class="label">Title</div>
            <div class="field" tal:content="context/title" />
        </div>

        <div class="row">
```

```
13            <div class="label">Author</div>
14            <div class="field" tal:content="view/author"/>
15        </div>
16
17        <div class="row">
18            <div class="label">Date/Time</div>
19            <div class="field" tal:content="view/modified"/>
20        </div>
21
22        <div class="row">
23            <div class="label">Parent</div>
24            <div class="field" tal:define="info view/parent_info">
25              <a href="../details.html"
26                  tal:condition="info"
27                  tal:content="info/title" />
28            </div>
29        </div>
30
31        <div class="row">
32            <div class="label">Body</div>
33            <div class="field" tal:content="context/body"/>
34        </div>
35
36    </div>
37  </body>
38 </html>
```

- Line 1–3 & 36–38: This is some standard boilerplate for a Zope page template that will embed the displayed data inside the common Zope 3 UI. This will ensure that all of the pages have a consistent look and feel to them and it allows the developer to concentrate on the functional parts of the view.
- Line 9: The `title` can be directly retrieved from the content object (the `Message` instance), which is available as `context`.
- Line 14 & 19: The author and the modification date/time are not directly available, since they are part of the object's meta data (Dublin Core). Therefore we need to make them available via the Python-based view class, which is provided to the template under the name `view`. A Python-based view class' sole purpose is to retrieve and prepare data to be in a displayable format.
- Line 24–27: While we probably could get to the parent via a relatively simple TALES path expression, it is custom in Zope 3 to make this the responsibility of the view class, so that the template contains as little logic as possible. In the next step you will see how this information is collected.

14.1.2 (b) Create the Python-based View class

From part (a) we know that we need the following methods (or attributes/properties) in our view class: `author()`, `modified()`, and `parent_info()`. First, create a

new file called `message.py` in the `browser` package. Note that we will place all browser-related Python code for `IMessage` in this module.

Here is the listing of my implementation:

```
1  from zope.app import zapi
2  from zope.app.dublincore.interfaces import ICMFDublinCore
3
4  from book.messageboard.interfaces import IMessage
5
6
7  class MessageDetails:
8
9      def author(self):
10         """Get user who last modified the Message."""
11         creators = ICMFDublinCore(self.context).creators
12         if not creators:
13             return 'unknown'
14         return creators[0]
15
16     def modified(self):
17         """Get last modification date."""
18         date = ICMFDublinCore(self.context).modified
19         if date is None:
20             date = ICMFDublinCore(self.context).created
21         if date is None:
22             return ''
23         return date.strftime('%d/%m/%Y %H:%M:%S')
24
25     def parent_info(self):
26         """Get the parent of the message"""
27         parent = zapi.getParent(self.context)
28         if not IMessage.providedBy(parent):
29             return None
30         return {'name': zapi.name(parent), 'title': parent.title}
```

▷ Line 1: Many of the fundamental utilties that you need, are available via the `zapi` module. The `zapi` module provides all crucial component architecture methods, such as `getParent()`. All the core servicenames are also available. Furthermore you can access traversal utilities as well. See `ZOPE3/src/zope/app/interfaces/zapi.py` for a complete list of available methods via the `zapi` module.

▷ Line 2: The `ICMFDublinCore` interface is used to store the Dublin Core meta data. Using this interface we can get to the desired information.

▷ Line 7: Note that the view class has no base class or specifies any implementing interface. The reason for this is that the ZCML directive will take care of this later on, by adding the `BrowserView` class as a base class of the view.

In some parts of Zope 3 you might still see the view class to inherit from `BrowserView`.

▷ Line 12–16: The code tries to get a list of `creators` (which I refer to as `authors`) from the Dublin Core meta data. If no `creator` is found, return the string

"unknown", otherwise the first `creator` in the list should be returned, which is the owner or the original author of the object. Note that we should usually have only one entry, since Messages are not edited (as of this stage of development).

- Line 20–28: Finding the modification date is a bit more tricky, since during the creation only the `created` field is populated and not the `modified` field. Therefore we try first to grab the `modified` field and if this fails we get the `created` field. If the `created` date/time does not exist, we return an empty string.

  Finally, if a date object was found, then we convert it to a string and return it.

- Line 30–33: Getting the parent is easy, just use the `getParent()` method. But then we need to make sure that the parent is also an `IMessage` object; if it is not, then we have a root message, and we return `None`. The `name` and the `title` of the parent are stored in an information dictionary, so that the data can be easily retrieved in a page template.

14.1.3 (c) Registering the View

The final task is to register the new view using ZCML. Open the configuration file in the `browser` sub-package and add the following lines:

```
1 <page
2     name="details.html"
3     for="book.messageboard.interfaces.IMessage"
4     class=".message.MessageDetails"
5     template="details.pt"
6     permission="zope.Public"
7     menu="zmi_views" title="Preview"/>
```

- Line 1: The `browser:page` directive registers a single page view.

- Line 2: The `name` attribute specifies the name as which the view will be accessible in the URL:
  `http://localhost:8080/board/message1/@@details.html`

  The `name` attribute is required.

- Line 3: The `for` attribute tells the system that this view is for `IMessage` objects. If this attribute is not specified, the view will be registered for `Interface`, which means for all objects.

- Line 4–5: Use the just created `MessageDetails` class and `details.pt` page template for the view; for this page `details.pt` will be rendered and uses an instance of `MessageDetails` as its `view`.

  Note that not all views need a supporting view class; therefore the `class` attribute is optional.

While you will usually specify a page template for regular pages, there are situations, where you would prefer a view on an attribute of the Python view class. In these cases you can specify the `attribute` attribute instead of `template`. The specified attribute/method should return a unicode string that is used as the final output.

- Line 6: The `permission` attribute specifies the permission that is required to see this page. At this stage we want to open up the details pages to any user of the site, so we assign the `zope.Public` permission, which is special, since every user, whether authenticated or not, has this permission.

- Line 7: In order to save ourselves from a separate menu entry directive, we can use the `menu` and `title` attribute to tell the system under which menu the page will be available. In this case, make it a tab (`zmi_views` menu) which will be called "Preview".

All you need to do now is to restart Zope, add a `Message` content object (if you have not done so yet) and click on it. The "Preview" tab should be available now. Note that you will have no "Parent" entry, since the message is not inside another one.

To see a "Parent" entry, add another message inside the current message by using the "Contents" view. Once you added the new message, click on it and go to the `Details` view. You should now see a "Parent" entry with a link back to the parent message.

14.1.4 (d) Testing the View

Before moving to the next item on the list, we should develop some functional tests to ensure that the view works correctly. Functional tests are usually straight forward, since they resemble the steps a user would take with the UI. The only possibly tricky part is to get all the form variables set correctly.

To run the functional tests the entire Zope 3 system is brought up, so that all side-effects and behavoir of an object inside its natural environment can be tested. Oftentimes very simple tests will suffice to determine bugs in the UI and also in ZCML, since all of it will be executed during the functional test startup.

The following functional tests will ensure that messages can be properly added and that the all the message details information are displayed in the "Preview". By convention all functional tests are stored in a sub-module called `ftests`. Since we plan to write many of these tests, let's make this module a package by creating the directory and adding an `__init__.py` file.

Now create a file called `test_message.py` and add the following testing code:

```
1  import unittest
2  from zope.app.tests.functional import BrowserTestCase
3
4  class MessageTest(BrowserTestCase):
5
6      def testAddMessage(self):
7          response = self.publish(
8              '/+/AddMessageBoard.html=board',
9              basic='mgr:mgrpw',
10             form={'field.description': u'Message Board',
11                   'UPDATE_SUBMIT': 'Add'})
12         self.assertEqual(response.getStatus(), 302)
13         self.assertEqual(response.getHeader('Location'),
14                          'http://localhost/@@contents.html')
15         response = self.publish(
16             '/board/+/AddMessage.html=msg1',
17             basic='mgr:mgrpw',
18             form={'field.title': u'Message 1',
19                   'field.body': u'Body',
20                   'UPDATE_SUBMIT': 'Add'})
21         self.assertEqual(response.getStatus(), 302)
22         self.assertEqual(response.getHeader('Location'),
23                          'http://localhost/board/@@contents.html')
24
25     def testMessageDetails(self):
26         self.testAddMessage()
27         response = self.publish('/board/msg1/@@details.html',
28                                 basic='mgr:mgrpw')
29         body = response.getBody()
30         self.checkForBrokenLinks(body, '/board/msg1/@@details.html',
31                                  basic='mgr:mgrpw')
32
33         self.assert_(body.find('Message Details') > 0)
34         self.assert_(body.find('Message 1') > 0)
35         self.assert_(body.find('Body') > 0)
36
37
38 def test_suite():
39     return unittest.TestSuite((
40         unittest.makeSuite(MessageTest),
41         ))
42
43 if __name__ == '__main__':
44     unittest.main(defaultTest='test_suite')
```

▷ Line 2: In order to simplify writing browser-based functional tests, the `BrowserTestCase` can be used as a test case base class. The most important convenience methods are used in the code below.

▷ Line 6–23: Before we are able to test views on a message, we have to create one. While it is possible to create a message using a lower-level API, this is a perfect chance to write tests for the adding views as well.

1. Line 7–11: The `publish()` method is used to publish a request with the publisher. The first arguments is the URL (excluding the server and port) to

be published. Commonly we also include the `basic` argument, which specifies the username and password. The system knows only about the user `zope.mgr` with username "mgr" and password "mgrpw". The role `zope.Manager` has been granted for this user, so that all possible screens should be availale.

In the `form` you specify a dictionary of all variables that are submitted via the HTTP form mechanism. The values of the entries can be already formatted Python objects and do not have to be just raw unicode strings. Note that the adding view requires a field named `UPDATE_SUBMIT` for the object to be added. Otherwise it just thinks this is a form reload.

2. Line 12–14: The adding view always returns a redirect (HTTP code 302). We can also verify the destination by looking at the "Location" HTTP header.

3. Line 15–23: Here we repeat the same procedure; this time by adding a message named "msg1" to the message board.

▷ Line 25–35: After creating the message object (line 26), the details view is simply requested and the HTML result stored in `body` (line 27–29).

One of the nice features of the `BrowserTestCase` is a method called `checkForBrokenLinks()` that parses the HTML looking for local URLs and then tries to verify that they are good links. The second argument of the method is the URL of the page that generated the body. This is needed to determine the location correctly. We should also specify the same authentication parameters, as used during the publication process, since certain links are only available if the user has the permission to access the linked page.

In the last the tests (line 33–35) we simply check that some of the expected information is somewhere in the HTML, which is usally efficient, since a faulty view usually causes a failure during the publishing process.

▷ Line 38–44: As always, we have to have the usual boilerplate.

Now that the tests have been developed, we can run them like the unit tests, except that for using the `-u` option (unit tests only), we now specify the `-f` option (functional tests only).

```
python2.3 test.py -vpf --dir src/book/messageboard
```

Since you already looked at the pages before, all tests should pass easily, unless you have a typo in your test case. Once the tests pass, feel free to go on to the next task.

14.2 Step II: Specifying the Default View

If you try to view a message using `http://localhost:8080/board/msg1` at this point, you will get the standard container `index.html` view. This is rather undesirable, since you default view should really show the contents of the message.

There is a special directive for declaring a default view. All you need to add are the following lines to your `browser` package configuration file:

```
1 <defaultView
2     for="book.messageboard.interfaces.IMessage"
3     name="details.html"/>
```

- Line 2: Here we tell the system that we are adding a default view for the components implementing `IMessage`.
- Line 3: We make the "Preview" screen the default view. However, you can choose whatever view you like. Naturally, these views are usually views that display data instead of asking for input. It is also advisable to make the least restrictive and most general view the default, so that users with only a few permissions can see something about the object.

14.3 Step III: Threaded Sub-Tree View

Creating a nice and extensible thread view is difficult, since the problem is recursive in nature. We would also like to have all HTML generation in Page Templates, since it allows us to enhance the functionality of the view later; however, Page Templates do not like recursion.

14.3.1 (a) Main Thread Page Template

So let's tackle the problem by starting to create the main view template for `thread.html`, which we call `thread.pt`:

```
1  <html metal:use-macro="views/standard_macros/view">
2    <body>
3      <div metal:fill-slot="body">
4
5        <h1>Discussion Thread</h1>
6
7        <div tal:replace="structure view/subthread" />
8
9      </div>
10   </body>
11 </html>
```

Almost everything is boiler plate really, but there is enough opportunity here to add some more functionality later, if we desire to do so.

▷ Line 7: Being blind about implementation, we simply assume that the Python-based view class will have a `subthread()` that can magically generate the desired sub-thread for this message or even the message board.

14.3.2 (b) Thread Python View Class

Next we have to build our Python view class. We start by editing a file called `thread.py` and insert the following code:

```
1  from zope.app.pagetemplate.viewpagetemplatefile import ViewPageTemplateFile
2  from book.messageboard.interfaces import IMessage
3
4  class Thread:
5
6      def __init__(self, context, request, base_url=''):
7          self.context = context
8          self.request = request
9          self.base_url = base_url
10
11     def listContentInfo(self):
12         children = []
13         for name, child in self.context.items():
14             if IMessage.providedBy(child):
15                 info = {}
16                 info['title'] = child.title
17                 url = self.base_url + name + '/'
18                 info['url'] = url + '@@thread.html'
19                 thread = Thread(child, self.request, url)
20                 info['thread'] = thread.subthread()
21                 children.append(info)
22         return children
23
24     subthread = ViewPageTemplateFile('subthread.pt')
```

▷ Line 1: The `ViewPageTemplateFile` class is used to allow page templates to be attributes/methods of a Python class. Very handy.

▷ Line 2: Import the `IMessage` interface, since we need it for object identification later.

▷ Line 25: Here is our promised `subthread()` method, which is simply a page template that knows how to render the thread. Note: You might want to read part (c) first, before proceeding.

▷ Line 12–23: This method provides all the necessary information to the subthread page template to do its work. For each child it generates an `info` dictionary. The interesting elements of the dictionary include the `url` and the `thread` values. The URL is built up in every iteration of the recursive process. We could also use the `zope.app.traversing` framework to generate the URL, but I think this is a much simpler this way.

The second interesting component of the info, the `thread` value, should contain a string with the HTML describing the subthread. This is were the recursion comes in. First we create a `Thread` instance (view) for each child. Then we are asking the view to return the subthread of the child, which is certainly one level deeper, which in return creates deeper levels and so on. Therefore the `thread` value will contain a threaded HTML representation of the branch.

14.3.3 (c) Sub-Thread Page Template

This template, named `subthread.pt` as required by the view class, is only responsible of creating an HTML presentation of the nested Message children using the information provided; therefore the template is very simple (since it contains no logic):

```
1 <ul>
2   <li tal:repeat="item view/listContentInfo">
3     <a href=""
4         tal:attributes="href item/url"
5         tal:content="item/title">Message 1</a>
6     <div tal:replace="structure item/thread"/>
7   </li>
8 </ul>
```

- ▷ Line 1 & 8: Unordered lists are always good to create threads or trees.
- ▷ Line 2: Thanks to the `Thread` view class, we simply need to iterate over the children information.
- ▷ Line 3–5: Make sure we show the title of the message and link it to the actual object.
- ▷ Line 6: Insert the subthread for the message.

14.3.4 (d) Register the Thread View

Registering the thread view works like before:

```
1 <page
2     name="thread.html"
3     for="book.messageboard.interfaces.IMessage"
4     class=".thread.Thread"
5     template="thread.pt"
6     permission="zope.View"
7     menu="zmi_views" title="Thread"/>
```

You should be familiar with the `page` directive already, so the above code should be easy to understand.

You also have to register the same view for `IMessageBoard`, so that you can get the full thread of the entire messageboard as well.

14.3.5 (e) Message Board Default View

Since the message board does not have a default view yet, let's make the thread view the default:

```
<defaultView
    for="book.messageboard.interfaces.IMessageBoard"
    name="thread.html"/>
```

This is, of course, very similar to the default view we registered for `IMessage` before.

14.4 Step IV: Adding Icons

Now that we have some text-based views, let's look into registering custom icons for the message board and message. Icons are also just views on objects, in this case our content components. However, to make life easier the `browser` namespace provides a convenience directive called `icon` to register icons.

Simply add the following directive for each content type in the `browser` package configuration file:

```
<icon
    name="zmi_icon"
    for="book.messageboard.interfaces.IMessage"
    file="message.png" />
```

The code should be self-explanatory at this point. Instead of a template, we are specifying a file here as the view, which is expected to be binary image data and not just ASCII text.

Now you should be all set. Restart Zope 3 and see whether the new features are working as expected.

The code is available in the Zope SVN under `http://svn.zope.org/book/trunk/messageboard/step02`.

Exercises

1. For the message details screen it might be also useful to display the author of the parent message. Expand the returned information dictionary of `parent_info` to include the author of the parent and display it properly using the template.
2. It would be great if there was a `Reply`, `Modify`, and `Delete` link (maybe as an image) behind each message title and make the actions work. Note that you should be able to reuse a lot of existing code for this.

CHAPTER 15

CUSTOM SCHEMA FIELDS AND FORM WIDGETS

Difficulty

Sprinter

Skills

- Be familiar with the results achieved in the previous two chapters.
- You should be comfortable with presentation components (views) as introduced in the previous chapter.

Problem/Task

So far we have created fairly respectable content components and some nice views for them. Let's now look at the fine print; currently it is possible that anything can be written into the message fields, including malicious HTML and Javascript. Therefore it would be useful to develop a special field (and corresponding widget) that strips out disallowed HTML tags.

Solution

Creating custom fields and widgets is a common task for end-user applications, since these systems have often very specific requirements. It was a design goal of the schema/form sub-system to be as customizable as possible, so it should be no surprise that it is very easy to write your own field and widget.

15.1 Step I: Creating the Field

The goal of the special field should be to verify input based on allowed or forbidden HTML tags. If the message body contains HTML tags other than the ones allowed or contains any forbidden tags, then the validation of the value should fail. Note that only one of the two attributes can be specified at once.

It is often not necessary to write a field from scratch, since Zope 3 ships with a respectable collection already. These serve commonly also as base classes for custom fields. For our `HTML` field the `Text` field seems to be the most appropriate base, since it provides most of the functionality for us already.

We will extend the `Text` field by two new attributes called `allowed_tags` and `forbidden_tags`. Then we are going to modify the `_validate()` method to reflect the constraint made by the two new attributes.

15.1.1 (a) Interface

As always, the first step is to define the interface. In the `messageboard`'s `interfaces` module, add the following lines:

```
1  from zope.schema import Tuple
2  from zope.schema.interfaces import IText
3
4  class IHTML(IText):
5      """A text field that handles HTML input."""
6
7      allowed_tags = Tuple(
8          title=u"Allowed HTML Tags",
9          description=u"""\
10         Only listed tags can be used in the value of the field.
11         """,
12         required=False)
13
14     forbidden_tags = Tuple(
15         title=u"Forbidden HTML Tags",
16         description=u"""\
17         Listed tags cannot be used in the value of the field.
18         """,
19         required=False)
```

▷ Line 1: The `Tuple` field simply requires a value to be a Python tuple.

▷ Line 2 & 4: We simple extend the `IText` interface and schema.

▷ Line 7–12 & 14–19: Define the two additional attributes using the field `Tuple`.

15.1.2 (b) Implementation

As previously mentioned, we will use the `Text` field as base class, since it provides most of the functionality we need. The main task of the implementation is to rewrite the validation method.

Let's start by editing a file called `fields.py` in the `messageboard` package and inserting the following code:

```
1  import re
2
3  from zope.schema import Text
4  from zope.schema.interfaces import ValidationError
5
6  forbidden_regex = r'</?(?:%s).*?/?>'
7  allowed_regex = r'</??(?!%s[ />])[a-zA-Z0-9]*? ?(?:[a-z0-9]*?="".*?")*/??>'
8
9  class ForbiddenTags(ValidationError):
10     __doc__ = u"""Forbidden HTML Tags used."""
11
12
13 class HTML(Text):
14
15     allowed_tags = ()
16     forbidden_tags = ()
17
18     def __init__(self, allowed_tags=(), forbidden_tags=(), **kw):
19         self.allowed_tags = allowed_tags
20         self.forbidden_tags = forbidden_tags
21         super(HTML, self).__init__(**kw)
22
23     def _validate(self, value):
24         super(HTML, self)._validate(value)
25
26         if self.forbidden_tags:
27             regex = forbidden_regex %'|'.join(self.forbidden_tags)
28             if re.findall(regex, value):
29                 raise ForbiddenTags(value, self.forbidden_tags)
30
31         if self.allowed_tags:
32             regex = allowed_regex %'[ />]|'.join(self.allowed_tags)
33             if re.findall(regex, value):
34                 raise ForbiddenTags(value, self.allowed_tags)
```

- ▷ Line 1: Import the Regular Expression module (`re`); we will use regular expressions to do the validation of the HTML.
- ▷ Line 3: Import the `Text` field that we will use as base class for the `HTML` field.
- ▷ Line 4 & 10–11: The validation method of the new HTML field will be able to throw a new type of validation error when an illegal HTML tag is found.

  Usually errors are defined in the `interfaces` module, but since it would cause a recursive import between the `interfaces` and `fields` module, we define it here.

- Line 7–9: These strings define the regular expression templates for detecting forbidden or allowed HTML tags, respectively. Note that these regular expressions are quiet more restrictive than what the HTML 4.01 standard requires, but it is good enough as demonstration. See exercise 1 at the end of the chapter to see how it should be done correctly.
- Line 16–19: In the constructor we are extracting the two new arguments and send the rest to the constructor of the `Text` field (line 21).
- Line 22: First we delegate validation to the `Text` field. The validation process might already fail at this point, so that further validation becomes unnecessary.
- Line 24–27: If forbidden tags were specified, then we try to detect them. If one is found, a `ForbiddenTags` error is raised attaching the faulty value and the tuple of forbidden tags to the exception.
- Line 29–32: Similarly to the previous block, this block checks that all used tags are in the collection of `allowed_tags` otherwise a `ForbiddenTags` error is raised.

We have an `HTML` field, but it does not implement `IHTML` interface. Why not? It is due to the fact that it would cause a recursive import once we use the HTML field in our content objects. To make the interface assertion, add the following lines to the `interfaces.py` module:

```
1 from zope.interface import classImplements
2 from fields import HTML
3 classImplements(HTML, IHTML)
```

At this point we should have a working field, but let's write some unit tests to verify the implementation.

15.1.3 (c) Unit Tests

Since we will use the `Text` field as a base class, we can also reuse the `Text` field's tests. Other than that, we simply have to test the new validation behavior.

In `messageboard/tests` add a file `test_fields.py` and add the following base tests. Note that the code is not complete (abbreviated sections are marked by `...`). You can find it in the source repository though.

```
1 import unittest
2 from zope.schema.tests.test_strfield import TextTest
3
4 from book.messageboard.fields import HTML, ForbiddenTags
5
6 class HTMLTest(TextTest):
7
8     _Field_Factory = HTML
9
```

```
10      def test_AllowedTagsHTMLValidate(self):
11          html = self._Field_Factory(allowed_tags=('h1','pre'))
12          html.validate(u'<h1>Blah</h1>')
13          ...
14          self.assertRaises(ForbiddenTags, html.validate,
15                            u'<h2>Foo</h2>')
16          ...
17
18      def test_ForbiddenTagsHTMLValidate(self):
19          html = self._Field_Factory(forbidden_tags=('h2','pre'))
20          html.validate(u'<h1>Blah</h1>')
21          ...
22          self.assertRaises(ForbiddenTags, html.validate,
23                            u'<h2>Foo</h2>')
24          ...
25
26  def test_suite():
27      return unittest.TestSuite((
28          unittest.makeSuite(HTMLTest),
29          ))
30
31  if __name__ == '__main__':
32      unittest.main(defaultTest='test_suite')
```

▷ Line 2: Since we use the `Text` field as base class, we can also use it's test case as base, getting some freebie tests in return.

▷ Line 8: However, the `TextTest` base comes with some rules we have to abide to. Specifying this `_Field_Factory` attribute is required, so that the correct field is tested.

▷ Line 10–16: These are tests of the validation method using the `allowed_tags` attribute. Some text was removed some to conserve space. You can look at the code for the full test suite.

▷ Line 18–24: Here we are testing the validation method using the `forbidden_tags` attribute.

15.2 Step II: Creating the Widget

Widgets are simply views of a field. Therefore we place the widget code in the `browser` sub-package.

Our `HTMLSourceWidget` will use the `TextAreaWidget` as a base and only the converter method `_convert(value)` has to be reimplemented, so that it will remove any undesired tags from the input value (yes, this means that the validation of values coming through these widgets will always pass.)

15.2.1 (a) Implementation

Since there is no need to create a new interface, we can start right away with the implementation. We get started by adding a file called `widgets.py` and inserting the following content:

```
1  import re
2  from zope.app.form.browser import TextAreaWidget
3  from book.messageboard.fields import forbidden_regex, allowed_regex
4
5  class HTMLSourceWidget(TextAreaWidget):
6
7    def _toFieldValue(self, input):
8        input = super(HTMLSourceWidget, self)._toFieldValue(input)
9
10       if self.context.forbidden_tags:
11           regex = forbidden_regex %'|'.join(
12               self.context.forbidden_tags)
13           input = re.sub(regex, '', input)
14
15       if self.context.allowed_tags:
16           regex = allowed_regex %'[ />]|'.join(
17               self.context.allowed_tags)
18           input = re.sub(regex, '', input)
19
20       return input
```

- ▷ Line 2: As mentioned above, we are going to use the `TextAreaWidget` as a base class.
- ▷ Line 3: There is no need to redefine the regular expressions for finding forbidden and non-allowed tags again, so we use the field's definitions. This will also avoid that the widget converter and field validator get out of sync.
- ▷ Line 8: We still want to use the original conversion, since it takes care of weird line endings and some other routine cleanups.
- ▷ Line 10–13: If we find a forbidden tag, simply remove it by replacing it with an empty string. Notice how we get the `forbidden_tags` attribute from the context (which is the field itself) of the widget.
- ▷ Line 15–18: If we find a tag that is not in the allowed tags tuple, then remove it as well.

Overall, this a very nice and compact way of converting the input value.

15.2.2 (b) Unit Tests

While we usually do not write *unit* tests for high-level view code, widget code should be tested, particularly the converter. Open `test_widgets.py` in `browser/tests` and insert:

```
1  import unittest
2  from zope.app.form.browser.tests.test_textareawidget import TextAreaWidgetTest
3  from book.messageboard.browser.widgets import HTMLSourceWidget
4  from book.messageboard.fields import HTML
5
6  class HTMLSourceWidgetTest(TextAreaWidgetTest):
7
8      _FieldFactory = HTML
9      _WidgetFactory = HTMLSourceWidget
10
11
12     def test_AllowedTagsConvert(self):
13         widget = self._widget
14         widget.context.allowed_tags=('h1','pre')
15         self.assertEqual(u'<h1>Blah</h1>',
16                          widget._toFieldValue(u'<h1>Blah</h1>'))
17         ...
18         self.assertEqual(u'Blah',
19                          widget._toFieldValue(u'<h2>Blah</h2>'))
20         ...
21
22     def test_ForbiddenTagsConvert(self):
23         widget = self._widget
24         widget.context.forbidden_tags=('h2','pre')
25
26         self.assertEqual(u'<h1>Blah</h1>',
27                          widget._toFieldValue(u'<h1>Blah</h1>'))
28         ...
29         self.assertEqual(u'Blah',
30                          widget._toFieldValue(u'<h2>Blah</h2>'))
31         ...
32
33 def test_suite():
34     return unittest.TestSuite((
35         unittest.makeSuite(HTMLSourceWidgetTest),
36         ))
37
38 if __name__ == '__main__':
39     unittest.main(defaultTest='test_suite')
```

▷ Line 2: Of course we are reusing the `TextAreaWidgetTest` to get some freebie tests.

▷ Line 8–9: Fulfilling the requirements of the `TextAreaWidgetTest`, we need to specify the field and widget we are using, which makes sense, since the widget must have the field (context) in order to fulfill all its duties.

▷ Line 12–31: Similar in nature to the field tests, the converter is tested. In this case however, we compare the output, since it can differ from the input based on whether forbidden tags were found or not.

15.3 Step III: Using the HTML Field

Now we have all the pieces we need. All that's left is to integrate them with the rest of the package. There are a couple of steps involved. First we register the `HTMLSourceWidget` as a widget for the `HTML` field. Next we need to change the `IMessage` interface declaration to use the `HTML` field.

15.3.1 (a) Registering the Widget

To register the new widget as a view for the `HTML` field we use the `zope` namespace `view` directive. Therefore you have to add the `zope` namespace to the configuration file's namespace list by adding the following line int he opening `configure` element:

```
1 xmlns:zope="http://namespaces.zope.org/zope"
```

Now add the following directive:

```
1 <zope:view
2     type="zope.publisher.interfaces.browser.IBrowserRequest"
3     for="book.messageboard.interfaces.IHTML"
4     provides="zope.app.form.interfaces.IInputWidget"
5     factory=".widgets.HTMLSourceWidget"
6     permission="zope.Public"
7     />
```

- ▷ Line 2: Since the `zope:view` directive can be used for any presentation type (for example: HTTP, WebDAV and FTP), it is necessary to state that the registered widget is for browsers (i.e. HTML).
- ▷ Line 3: This widget will work for all fields implementing `IHTML`.
- ▷ Line 4: In general presentation component, like adapters, can have a specific output interface. Usually this interface is just `zope.interface.Interface`, but here we specifically want to say that this is a widget that is accepting *input* for the field. The other type of widget is the `DisplayWidget`.
- ▷ Line 5: Specifies the factory or class that will be used to generate the widget.
- ▷ Line 6: We make this widget publically available, meaning that everyone using the system can use the widget as well.

15.3.2 (b) Adjusting the `IMessage` interface

The final step is to use the field in the `IMessage` interface. Let's go to the `interfaces` module to decide which property is going to become an `HTML` field. The field is already imported.

Now, we definitely want to make the `body` property of `IMessage` an HTML field. We could also do this for `description` of `IMessageBoard`, but let's not to do that for reasons of keeping it simple. So here are the changes that need to be done to the `body` property declaration (starting at line 24):

```
1  body = HTML(
2      title=u"Message Body",
3      description=u"This is the actual message. Type whatever!",
4      default=u"",
5      allowed_tags=('h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'img', 'a',
6                    'br', 'b', 'i', 'u', 'em', 'sub', 'sup',
7                    'table', 'tr', 'td', 'th', 'code', 'pre',
8                    'center', 'div', 'span', 'p', 'font', 'ol',
9                    'ul', 'li', 'q', 's', 'strong'),
10     required=False)
```

▷ Line 5–9: Here is our new attribute that was added in the `IHTML` interface. This is my choice of valid tags, so feel free to add or remove whatever tags you like.

And that's it! You are done. To try the result of your work, restart Zope 3, start editing a new message and see if it will accept tags like `html` or `body`. You should notice that these tags will be silently removed from the message body upon saving it.

Exercises

1. Instead of using our own premature HTML cleanup facilities, we really should make use of Chris Wither's HTML Strip-o-Gram package which can be found at `http://www.zope.org/Members/chrisw/StripOGram`. Implement a version of the `HTML` field and `HTMLSourceWidget` widget using this package.

2. Sometimes it might be nice to also allow HTML for the title of the messages, therefore you will also need an HTML version for the `TextLine` field and the `TextWidget`. Abstract the current converter and validation implementation, so that it is usable for both, message `title` and `body`.

3. Using only HTML as input can be boring and tedious for some message board applications. In the zwiki for Zope 3 packge we make use of a system (`zope.app.renderer`) that let's you select the type of input and then knows how to render each type of input for the browser. Insert this type of system into the message board application and merge it with the HTML validation and conversion code.

CHAPTER 16

SECURING COMPONENTS

Difficulty

Sprinter

Skills

- Be Knowledgable about topics covered in the previous chapters of the "Content Components" section.
- Be familiar with interfaces, ZCML and some of the security concepts, such as permissions, roles and principals.

Problem/Task

While we had to make basic security assertions in order to get our message board to work, it does not really represent a secure system at this point. Some end-user views require the `zope.ManageContent` permission and there are no granular roles defined.

Solution

Zope 3 comes with a flexible security mechanism. The two fundamental concepts are permissions and principals. Permissions are like keys to doors that open to a particular functionality. For example, we might need the permission `zope.View` to look at a message's detail screen. Principals, on the other hand, are agents of the system that execute actions. The most common example of a principal is a user of the system. The goal is now to grant permissions to principals, which is the duty of another sub-system known as the `securitypolicy`.

Zope 3 does not enforce any particular security policy. In contrary, it encourages site administrators to carefully choose the security policy and use one that fits their needs best. The default Zope 3 distribution comes with a default security policy (`zope.app.securitypolicy`) that supports the concept of roles. Roles are like hats people wear, as Jim Fulton would say, and can be seen as a collection of permissions. A single user can have several hats, but only wear one at a time. Prominent examples of roles include "editor" and "administrator". Therefore, the default security policy supports mappings from permissions to principals, permissions to roles, and roles to principals. This chapter will use the default security policy to setup the security, but will clearly mark the sections that are security policy specific.

The first task will be to define a sensible set of permissions and change the existing directives to use these new permissions. This is a bit tedious, but it is important that you do this carefully, since the quality of your security depends on this task. While doing this, you usually discover that you missed a permission and even a role, so do not hesitate to add some. That is everything the programmer should ever do. The site administrator, who uses the default security policy, will then define roles and grant permissions to them. Finally the roles are granted to some users for testing.

Securing an object does not require any modification to the existing Python code as you will see going through the chapter, since everything is configured via ZCML. Therefore security can be completely configured using ZCML, leaving the Python code untouched, which is another advantage of using Zope 3 (in comparison to Zope 2, for example).

16.1 Step I: Delcarations of Permissions

Other than in Zope 2, permissions have to be explicitly defined. For our message board it will suffice to define the following four basic permissions:

- `View` – Allow users to access the data for message boards and messages. Every regular message board `User` is going to have this permission.
- `Add` – Allows someone to create (i.e. post) a message and add it to the message board or another message. Note that every regular `User` is allowed to do this, since posting and replying must be possible.
- `Edit` – Editing content (after it is created) is only a permission the message board `Editor` possesses (for moderation), since we would not want a regular user to be able to manipulate posts after creation.
- `Delete` – The `Editor` must be able to get rid of messages, of course. Therefore the `Delete` permission is assigned to her. Note that this permission does not allow the editor to delete `MessageBoard` objects from folders or other containers.

Let's define the permissions now. Note that they must appear at the very beginning of the configuration file, so that they will de defined by the time the other directives (that will use the permissions) are executed. Here are the four directives you should add to your main `configure.zcml` file:

```
<permission
    id="book.messageboard.View"
    title="View Message Board and Messages"
    description="View the Message Board and all its content."
    />
<permission
    id="book.messageboard.Add"
    title="Add Message"
    description="Add Message."
    />
<permission
    id="book.messageboard.Edit"
    title="Edit Messages"
    description="Edit Messages."
    />
<permission
    id="book.messageboard.Delete"
    title="Delete Message"
    description="Delete Message."
    />
```

The `zope:permission` directive defines and creates a new permission in the global permission registry. The `id` should be a unique name for the permission, so it is a good idea to give the name a dotted prefix, like `book.messageboard.` in this case. Note that the `id` must be a valid URI or a dotted name – if there is no dot in the dotted version, a `ValidationError` will be raised. The `id` is used as identifier in the following configuration steps. The `title` of the permissions is a short description that will be used in GUIs to identify the permission, while the `description` is a longer explanation that serves more or less as documentation. Both the `id` and `title` are required attributes.

16.2 Step II: Using the Permissions

Now that we have defined these permissions, we also have to use them; let's start with the main message board configuration file (`messageboard/configure.zcml`). In the following walk-through we are only going to use the last part of the permission name to refer to the permission, leaving off `book.messageboard`. However, the full `id` has to be specified for the configuration to execute.

- Change the first require statement of in the `MessageBoard` content directive to use the `View` permission (line 42). This makes the `description` and the items accessible to all board users. Similarly, change line 64 for the `Message`.

- Change the permission of line 46 to `Edit`, since only the message board administrator should be able to change any of the properties of the `MessageBoard` object.
- All the container functionality will only require the view permission, so change the permission on line 68 to `View`. This is unsecure, since this includes read and write methods, but it will suffice for this demonstration.
- For the `Message` we need to be able to set the attributes with the `Add` permission, so change line 72 to specify this permission.

Now let's go to the `browser` configuration file (`messageboard/browser/configure.zcml`) and fix the permissions there.

- The permissions for the message board's add form (line 11), add menu item (line 18), and its edit form (line 27) stay unchanged, since only an administrator should be able manage the board.
- Since we want every user to see the messages in a messageboard, the permission on line 33 should become `View`. Since the `contents` view is meant for management, only principals with the `Edit` permission should be able to see it (line 34). Finally, you need the `Add` permission to actually add new messages to the message board (line 35). The same is true for the message's container views permissions (line 84–86).
- Since all user should be able to see the message thread and the message details, the permissions on line 43, 94, and 106 should become `View`.
- On line 61 you should change the permission to `Add`, because you only allow messages to be added to the message board, if the user has this permission. The same is true for the message's add menu item on line 68.
- On line 78 make sure that a user can only access the edit screen if he has the `Edit` permission.

That's it. If you would restart Zope 3 at this point, you could not even access the `MessageBoard` and/or `Message` instances. Therefore we need to create some roles next and assign permissions to them.

16.3 Step III: Declaration of Roles

The declaration of roles is specific to Zope 3's default security policy. Another security policy might not even have the concept of roles at all. Therefore, the role

declaration and grants to the permissions should not even be part of your package. For simplicity and keeping it all at one place, we are going to store the policy-specific security configuration in `security.zcml`. For our message board package we really only need two roles, "User" and "Editor", which are declared as follows:

```
<role
    id="book.messageboard.User"
    title="Message Board User"
    description="Users that actually use the Message Board."/>

<role
    id="book.messageboard.Editor"
    title="Message Board Editor"
    description="The Editor can edit and delete Messages."/>
```

Equivalently to the `zope:permission` directive, the `zope:role` directive creates and registers a new role with the global role registry. Again, the `id` must be a unique identifier that is used throughout the configuration process to identify the role. Both, the `id` and the `title` are required.

Next we grant the new permissions to the new roles, i.e. create a permission-role map. The user should be only to add and view messages, while the editor is allowed to execute all permission.

```
<grant
    permission="book.messageboard.View"
    role="book.messageboard.User"
    />
<grant
    permission="book.messageboard.Add"
    role="book.messageboard.User"
    />
<grant
    permission="book.messageboard.Edit"
    role="book.messageboard.Editor"
    />
<grant
    permission="book.messageboard.Delete"
    role="book.messageboard.Editor"
    />
```

The `zope:grant` directive is fairly complex, since it permits all three different types of security mappings. It allows you to assign a permission to a principal, a role to a principal, and a permission to a role. Therefore the directive has three optional arguments: `permission`, `role`, and `principal`. Exactly two of the three arguments have to be specified to make it a valid directive. All three security objects are specified by their `id`.

Finally, you have to include the `security.zcml` file into your other configuration. This is simply done by adding the following inclusion directive in the `ZOPE3/principals.zcml` file:

```
<include package="book.messageboard" file="security.zcml" />
```

The reason we put it here is to make it obvious that this file depends on the security policy. Also, when assigning permissions to roles we want all possible permissions the system can have to be defined. Since the `principals.zcml` file is the last ZCML to be evaluated, this is the best place to put the declarations.

16.4 Step IV: Assigning Roles to Principals

To make our package work again, we now have to connect the roles to some principals. We are going to create two new principals called `boarduser` and `boardeditor`. To do that, go to the Zope 3 root directory and add the following lines to `principals.zcml`:

```
<principal
    id="book.messageboard.boarduser"
    title="Message Board User"
    login="boarduser" password="book"
    />
<grant
    role="book.messageboard.User"
    principal="book.messageboard.boarduser"
    />

<principal
    id="book.messageboard.boardeditor"
    title="Message Board Editor"
    login="boardeditor" password="book"
    />
<grant
    role="book.messageboard.User"
    principal="book.messageboard.boardeditor"
    />
<grant
    role="book.messageboard.Editor"
    principal="book.messageboard.boardeditor"
    />
```

The `zope:principal` directive creates and registers a new principal/user in the system. Like for all security object directives, the `id` and `title` attributes are required. We could also specify a `description` as well. In addition to these three attributes, the developer *must* specify a login and password (plain text) for the user, which is used for authentication of course.

Note that you might want to grant the `book.messageboard.User` role to the `zope.anybody` principal, so that everyone can view and add messages.

The `zope.anybody` principal is an unauthenticated principal, which is defined using the `zope:unauthenticatedPrincipal` directive, which has the same three basic attributes the `zope:principal` directive had, but does not accept the `login` and `password` attribute.

Now your system should be secure and usable. If you restart Zope 3 now, you will see that only the message board's `Editor` can freely manipulate objects. (Of course you have to log in as one.)

Exercises

1. In retrospect it was a bad idea to give the `book.messageboard.User` role the `book.messageboard.View` and `book.messageboard.Add` permission, since you cannot differentiate between an anonymous user reading the board and a user being able to add messages anymore. Add yet another role called `book.messageboard.Viewer` and make the `details.html` and `thread.html` view available to this role. Then grant this role to the unauthenticated principal (`anybody`) and verify that this principal can indeed access these views.

2. (Referring to the previous problem) On the other hand, instead of creating another role, we could just grant the `View` permission to the anonymous principal directly. Do that and ensure that the unauthenticated principal can see these views.

CHAPTER 17

CHANGING SIZE INFORMATION

Difficulty

Newcomer

Skills

- You should be familiar with the previous chapters of the "Content Components" section.

Problem/Task

Currently, when looking at the contents view of a message, it will show you the amount of items in the message, which includes reply-messages and attachments (files and images). It would be nice if the size field would say "x replies, y attachments".

Solution

The size output is handled by a very simple adapter, which will adapt from `IMessage` to `ISized`.

17.1 Step I: Implementation of the Adapter

An adapter is usually a simple class, which is marked by the fact that it takes one object as constructor argument. This object must provide the "from" interface that is often also listed in the `__used_for__` attribute of the class. Add the following code in your `message.py` file:

```
1  from zope.app.size.interfaces import ISized
2
3  class MessageSized(object):
4
5      implements(ISized)
6      __used_for__ = IMessage
7
8      def __init__(self, message):
9          self._message = message
10
11     def sizeForSorting(self):
12         """See ISized"""
13         return ('item', len(self._message))
14
15     def sizeForDisplay(self):
16         """See ISized"""
17         messages = 0
18         for obj in self._message.values():
19             if IMessage.providedBy(obj):
20                 messages += 1
21
22         attachments = len(self._message)-messages
23
24         if messages == 1: size = u'1 reply'
25         else: size = u'%i replies' %messages
26
27         if attachments == 1: size += u', 1 attachment'
28         else: size += u', %i attachments' %attachments
29
30         return size
```

The `ISized` interface specifies two methods:

- ▷ Line 10–12: `sizeForString()` must return a tuple with the first element being a unit and the second the value. This format was chosen to provide a generic comparable representation of the size.
- ▷ Line 14–29: `sizeForDisplay()` can return any sort of unicode string that represents the size of the object in a meaningful way. The output should not be too long (mine is already very long). As promised it displays both responses and attachments separately.

17.2 Step II: Unit tests

Now let's write some doc tests for the adapter. Add the following tests In the doc string of the `sizeForSorting()` method:

```
1  Create the adapter first.
2
3  >>> size = MessageSized(Message())
4
5  Here are some examples of the expected output.
```

```
6
7 >>> size.sizeForSorting()
8 ('item', 0)
9 >>> size._message['msg1'] = Message()
10 >>> size.sizeForSorting()
11 ('item', 1)
12 >>> size._message['att1'] = object()
13 >>> size.sizeForSorting()
14 ('item', 2)
```

The test is straight forward, since we add an object and check whether it increased the size of items by one. In the `sizeForDisplay()` doc string add:

```
1 Create the adapter first.
2
3 >>> size = MessageSized(Message())
4
5 Here are some examples of the expected output.
6
7 >>> size.sizeForDisplay()
8 u'0 replies, 0 attachments'
9 >>> size._message['msg1'] = Message()
10 >>> size.sizeForDisplay()
11 u'1 reply, 0 attachments'
12 >>> size._message['msg2'] =  Message()
13 >>> size.sizeForDisplay()
14 u'2 replies, 0 attachments'
15 >>> size._message['att1'] = object()
16 >>> size.sizeForDisplay()
17 u'2 replies, 1 attachment'
18 >>> size._message['att2'] = object()
19 >>> size.sizeForDisplay()
20 u'2 replies, 2 attachments'
```

The doc tests are already registered, since the `message.py` file already contains some doc tests. However, adding an object to a container requires some of the component architecture to be up and running. There exists a testing convenience module called `zope.app.tests.placelesssetup`, which contains two functions `setUp()` and `tearDown()` that can be passed in the doc test suite as positional arguments. therefore the test suite declaration changes from

```
1 DocTestSuite('book.messageboard.message')
```

to

```
1 DocTestSuite('book.messageboard.message',
2              setUp=setUp, tearDown=tearDown)
```

You can now run the tests the usual way.

```
python2.3 test.py -vpu --dir src/book/messageboard
```

17.3 Step III: Registration

Now we register the adapter in `messageboard/configure.zcml` using the following ZCML directive:

```
<adapter
    factory=".message.MessageSized"
    provides="zope.app.size.interfaces.ISized"
    for=".interfaces.IMessage"
    />
```

The `zope:adapter` is the way to register global adapters via ZCML. The `factory` attribute allows you to specify a list of factories (usually only one is specified) that are responsible for creating an adapter instance that takes an object implementing the interface specified in the `for` attribute and providing the interface specified in `provides`. All of these three attributes are mandatory.

For our case, we basically say that an instance of the `MessageSized` class provides an `ISized` interface for objects implementing `IMessage`.

The directive also supports two optional arguments. We can also specify a `permission`. The adapter will be only available to the principal, if the principal has the specified permission. If no permission is specified, everyone can access the adapter. The other optional argument of the directive is the `name` attribute that specifies the name of the adapter. Using names, we can specify multiple adapters from one interface to another.

That's it! Restart Zope 3 and see for yourself. Note how we did not need to touch any existing Python code to provide this functionality.

Exercises

1. Write an `ISized` adapter for `IMessageBoard` that outputs `xmessages` as the displayable size.

CHAPTER 18

INTERNATIONALIZING A PACKAGE

Difficulty

Sprinter

Skills

- You should be familiar with the previous chapters of the "Content Components" section.
- Familiarity with Page Templates is desired.
- Basic knowledge of the gettext format and tools is also a plus. Optional.

Problem/Task

Now that we have a working message board package, it is time to think about our friends overseas and the fact that not everyone can speak English. Therefore it is our task now to internationalize and localize the code to . . . let's say German.

Solution

Before we can start coding, it is important to cover some of the basics. You might already have wondered about the difference between the terms internationalization and localization.

- Internationalization (I18n) is the process of making a package translatable, basically the programmer's task of inserting the necessary code so that human-readable strings can be translated and dates/times be formatted, respecting the users "locale" settings.

- Localization (L10n) is the process of actually creating a translation for a particular language. Often translations are not done by programmers but by translators (or formally, localization vendors).

But what is a so-called "locale"? Locales are objects that contain information about a particular physical/abstract region in the world, such as language, dialect, monetary unit, date/time/number formats and so on. An example of a locale would be "de_DE_PREEURO" (language, country/region, variant), which describes Germany before the Euro was introduced. However, "de" is also a valid locale, referring to all German speaking regions. So you can imagine that there is a locale hierarchy. "de_DE_PREEURO" is more specific than "de_DE", which is in turn more specific than "de". So if the user's locale setting is "de_DE_PREEURO" and we want to look for the date format template, the system will look up the path in "de_DE_PREEURO", then "de_DE" and finally in "de", where it will find it.

Note that this chapter has little to do with Python development, but is still useful to know, since all Zope 3 core components are required to be internationalized.

18.1 Step I: Internationalizing Python code

There should be only a few spots where internationalizing is necessary, since translatable strings are used for views, which are usually coded in Page Templates. One of the big exceptions are schemas, since we always define human readable titles, descriptions and default text values for the declared fields.

Zope uses message ids to mark strings as translatable. Translatable strings must always carry a domain, so that we know to which translation domain to pick.

We use message id factories to create message ids:

```
1 from zope.i18n import MessageIDFactory
2 _ = MessageIDFactory('messageboard')
```

▷ Line 1–2: Every Python file containing translatable strings must contain this small boiler plate. Note that for Zope 3 core code we have a short cut:

```
1 from zope.app.i18n import ZopeMessageIDFactory as _
```

This import creates a message id factory that uses the "zope" domain.

▷ Line 2: The underscore character is commonly used to represent the "translation function" (from gettext). In our case it is used as message id constructor/factory. The argument of the `MessageIDFactory` is the domain, which is in our case `messageboard`.

But why do we need domains in the first place? My favorite example for the need of domains is the word `Sun`. This word really represents three different meanings in English: (1) our star the Sun, (2) an abbreviation for Sunday and (3) the company Sun Microsystems. All of these meanings have different translations in German for example. So you can distinguish between them by specifying domains, such as "astronomy", "calendar" and "companies", respectively. Domains also allow us to organize and reuse translations; they are almost like libraries. For example, not every single package needs to collect its own "calendar" translations, but all packages could benefit from one cohesive domain.

Another way of categorizing translations is by creating somewhat abstract message strings. So for example the value of an add button becomes `add-button` instead of the usual `Add` and translations for this string would then insert the human readable string, such as `Add` for English or `Hinzufgen` for German. We will see this usage specifically in Page Templates (see next section). These "abstract message strings" are known as "explicit message ids".

You might also wonder why we have to use the message id concept, instead of using a translation function directly, like other desktop applications do. Here we should recall that Zope is an Application Server and has multiple users that are served over the network. So at the time a piece of code is called, we often do not know anything about the user or the desired language. Only views (be it Python code or Page Templates) have information about the user and therefore the desired locale, which contains the language, so that the translation has to be prolonged as long as possible. As a rule of thumb, I always say that translating is the last task the application should do before providing the final end-user output. Zope 3 honors this rule in every aspect.

But let's get back to translating Python code. Since the interfaces have the most translatable strings, we start with them. Open the `interfaces.py` module and add the above mentioned boiler plate. Now, we internationalize each field. For example, the `IMessageBoard` schema's `description` field is changed from

```
description = Text(
    title=u"Description",
    description=u"A detailed description of the content of the board.",
    default=u"",
    required=False)
```

to

```
description = Text(
    title=_("Description"),
    description=_("A detailed description of the content of the board."),
    default=u"",
    required=False)
```

Note how the underscore message id factory simply functions like a translating message. Do the same transformation for all schemas in the interfaces module. Also,

note that while `title` and `description` require unicode strings, we can simply pass a regular string into the message id factory, since the message id uses `unicode` as its base class, making the message id look like a `unicode` object. Another minor translation is the `__doc__` attribute of the `ForbiddenTags` class in the `fields` module. Make sure to internationalize this one as well in the same manner.

One more interesting case of marking message strings is found in `message.py` in the `MessageSized` class, `sizeForDisplay()` method. The original code was

```
1 if messages == 1: size = u'1 reply'
2 else: size = u'%i replies' %messages
3
4 if attachments == 1: size += u', 1 attachment'
5 else: size += u', %i attachments' %attachments
```

This usage causes a problem to our simplistic usage of message ids, since we now have variables in our string and something like `‘messages‘+_("replies")` will not work contrary to gettext applications, since the underscore object will not actually do the translation. The lookup for the translation would simply fail, since the system would look for translations like "2 replies", "3 replies" and so on. All this means is that the actual variable values need to be inserted into the text *after* the translation. For exactly this case, the `MessageId` object has a `mapping` attribute that can store all variables and will insert them after the translation is completed. This means of course that we also have to markup our text string in a different way, so that the new code becomes:

```
1  if messages == 1 and attachments == 1:
2      size = _('1 reply, 1 attachment')
3  elif messages == 1 and attachments != 1:
4      size = _('1 reply, ${attachments} attachments')
5  elif messages != 1 and attachments == 1:
6      size = _('${messages} replies, 1 attachment')
7  else:
8      size = _('${messages} replies, ${attachments} attachments')
9
10 size.mapping = {'messages': ‘messages‘, 'attachments': ‘attachments‘}
```

▷ 1–8: Here we handle the four different cases we could possibly have. While this might not be the most efficient way of doing it, it allows us to list all four combinations separately, so that the message string extraction tool will be able to find it. This tool looks for strings that are enclosed by `_()`.

Note how the `%i` occurrences were replaced by `${messages}` and `${attachments}`, which is the translation domain way of marking a later to be inserted variable.

▷ Line 10: Once the message id is constructed, we add the mapping with the two required variable values.

Since we have tests written for the size adapter, we need to correct them at this point as well. You might try to fix the tests yourself, before reading on. Change the doc string of the `sizeForDisplay()` method to

```
1  Creater the adapter first.
2
3  >>> size = MessageSized(Message())
4
5  Here are some examples of the expected output.
6
7  >>> str = size.sizeForDisplay()
8  >>> str
9  u'${messages} replies, ${attachments} attachments'
10 >>> 'msgs: %(messages)s, atts: %(attachments)s' %str.mapping
11 'msgs: 0, atts: 0'
12 >>> size._message['msg1'] = Message()
13 >>> str = size.sizeForDisplay()
14 >>> str
15 u'1 reply, ${attachments} attachments'
16 >>> 'msgs: %(messages)s, atts: %(attachments)s' %str.mapping
17 'msgs: 1, atts: 0'
18 >>> size._message['att1'] = object()
19 >>> str = size.sizeForDisplay()
20 >>> str
21 u'1 reply, 1 attachment'
22 >>> 'msgs: %(messages)s, atts: %(attachments)s' %str.mapping
23 'msgs: 1, atts: 1'
24 >>> size._message['msg2'] =  Message()
25 >>> str = size.sizeForDisplay()
26 >>> str
27 u'${messages} replies, 1 attachment'
28 >>> 'msgs: %(messages)s, atts: %(attachments)s' %str.mapping
29 'msgs: 2, atts: 1'
30 >>> size._message['att2'] = object()
31 >>> str = size.sizeForDisplay()
32 >>> str
33 u'${messages} replies, ${attachments} attachments'
34 >>> 'msgs: %(messages)s, atts: %(attachments)s' %str.mapping
35 'msgs: 2, atts: 2'
```

▷ Line 7–11: The `sizeForDisplay()` method now returns a message id object. The message id uses simply its text part for representation. In the following lines it is checked that the mapping exists and contains the correct values.

▷ Line 12–35: Repetion of the test as before using different amounts of replies and attachments.

One last location where we have to internationalize some Python code output is in `browser/message.py`. The string `'unknown'` must be wrapped in a message id factory call. Also, the string returned by the `modified()` method of the `MessageDetails` view class must be adapted to use the user's locale information, since it returns a formatted date/time string. Since `MessageDetails` is a *view* class, we have the user's locale available, so that we can change the old version

```
return date.strftime('%d/%m/%Y %H:%M:%S')
```

easily to the internationalized version

```
formatter = self.request.locale.dates.getFormatter('dateTime', 'short')
return formatter.format(date)
```

Every `BrowserRequest` instance has a `locale` object, which represents the user's regional settings. The `getFormatter()` method returns a formatter instance that can format a `datetime` object to a string based on the locale. Refer to the API reference to see all of the locale's functionality.

This is already everything that has to be done in the Python code. If you do not believe me, feel free to check the other Python modules for translatable strings – you will not find any. As promised, Python code contains only a few places of human readable strings and this is a good thing.

18.2 Step II: Internationalizing Page Templates

Internationalizing Page Templates is more interesting in many ways. We do not only have to worry about finding the correct tags to internationalize, but since we also can have heavy nesting, the complexity can become overwhelming. My suggestion: Keep the content of translatable tags as flat as possible, i.e. try to have translatable text that does not contain much HTML and TAL code.

To achieve internationalization support in Zope 3's Page Templates, we designed a new `i18n` namespace. It is well documented at `http://dev.zope.org/Zope3/ZPTInternationalizationSupport`. The three most common attributes are `i18n:domain`, `i18n:translate` and `i18n:attributes`. Note that the `i18n` namespace has been back-ported to Zope 2 as well, so you might be familiar with it already.

The cleanest Page Template in the `browser` package is `details.pt`, so let's internationalize it first:

```
<html metal:use-macro="views/standard_macros/page">
  <body>
    <div metal:fill-slot="body" i18n:domain="messageboard">

      <h1 i18n:translate="">Message Details</h1>

        <div class="row">
            <div class="label" i18n:translate="">Title</div>
            <div class="field" tal:content="context/title" />
        </div>

        <div class="row">
            <div class="label" i18n:translate="">Author</div>
            <div class="field" tal:content="view/author"/>
        </div>

        <div class="row">
```

```
18              <div class="label" i18n:translate="">Date/Time</div>
19              <div class="field" tal:content="view/modified"/>
20          </div>
21
22          <div class="row">
23              <div class="label" i18n:translate="">Parent</div>
24              <div class="field" tal:define="info view/parent_info">
25                <a href="../"
26                    tal:condition="info"
27                    tal:content="info/title" />
28              </div>
29          </div>
30
31          <div class="row">
32              <div class="label" i18n:translate="">Body</div>
33              <div class="field" tal:content="structure context/body"/>
34          </div>
35
36      </div>
37    </body>
38  </html>
```

▷ Line 3: The best place for the domain specification is this `div` tag, since it is inside a specific slot and will not influence other template files' domain settings.

▷ Line 8, 13, 18, 23 & 32: The `i18n:translate=""` just causes the content of the `div` tag to be translated.

Note that there was no need here to use `i18n:attributes`. However, when we deal with buttons, we use this instruction quite often. Here an example:

```
1 <input type="submit" value="Add" i18n:attributes="value add-button" />
```

Similar to `tal:attributes` the `value`-attribute value will be replaced by the translation of `add-button` or remains as the default string (`Add`), if no translation was found.

This is really everything that is needed for Page Templates. As exercise 1 and 2 state, you should finish the other templates yourself. (Hint: If you do exercise 1 at this point, you can skip exercise 2.)

18.3 Step III: Internationalizing ZCML

Internationalizing ZCML is a one time, one step process. All you need to do here is to add a `i18n_domain="messageboard"` attribute assignment in your `configure` tag of the main `configure.zcml` file. It is the responsibility of the directive author to specify which attribute values should be converted to message ids, so that you have to worry about nothing else. All of this might seem a bit magical, but it is

explicit and an incredibly powerful feature of Zope 3's translation system, since it minimizes the overhead of internationalizing ZCML code.

Setting this attribute will get rid of all the warnings you experienced until now when starting up Zope 3.

18.4 Step IV: Creating Language Directories

The directory structure of the translations has to follow a strict format, since we tried to keep it gettext compatible. By convention we keep all message catalogs and message catalog templates in a directory called `locales` (so create it now). This directory typically contains the message catalog template file (extension `pot`) and the various language directories, such as `en`. Since we want to create a translation for English and German, create the directories `en` and `de`, respectivily.

Now comes the part that might not make sense at first. The language directories do not contain the message catalog directly, but another directory called `LC_MESSAGES`. Create them in each language directory.

Since English is our default language, we always want it to use the default value. Therefore the message catalog can be totally empty (or just contain the meta-data). Create a file called `messageboard.po` in the `locales/en/LC_MESSAGES` directory and add the following comment and meta data.

```
# This file contains no message ids because the messageboard's default
# language is English
msgid ""
msgstr ""
"Project-Id-Version: messageboard\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=UTF-8\n"
"Content-Transfer-Encoding: 8bit\n"
```

Now you are done with the preparations. Before we can localize the message board, we need to create the message catalogs as it will be described in the next section.

18.5 Step V: Extracting Translatable Strings

Zope provides a very powerful extraction tool to grab all translatable text strings from Python, Page Template and ZCML files. With each translatable string, the file and line number is recorded and later added as comment in the message catalog template file.

After all strings were collected and duplicates merged into single entries, the tool saves the strings in a message catalog template file called ⟨`domain`⟩`.pot`. This is

the beginning of localization. From now on we are only concerned about using the template to create translations.

The extraction tool, called `i18nextract.py`, can be found in `ZOPE3/utilities`. Before executing the tool, add your Zope 3 source directory to the `PYTHONPATH`, so that all necessary modules are found. In `bash` the `PYTHONPATH` can be set using

```
export PYTHONPATH=$PYTHONOATH:ZOPE3/src
```

To execute the tool, go to the `messageboard` directory and enter the following command. Make sure that you entered the absolute path for `ZOPE3`, since the tool does not work well with symlinks.

```
python ZOPE3/utilities/i18nextract.py -d messageboard -p ./ -o ./locales
```

This will extract all translatable strings from the message board package and store the template file as `messageboard/locales/messageboard.pot`.

As you can see, the tool supports three options plus an help option:

- `-h/--help` – Print the help of the `i18nextract.py` tool on the screen and exit.
- `-d/--domain⟨domain⟩` – This option specifies the domain, in our case `messageboard`, that is supposed to be extracted.
- `-p/--path⟨path⟩` – The path specifies the package by path that is searched for translatable strings. In our case we just used `./`, since we already were in the package.
- `-odir` – This option specifies a directory, relative to the package in which to put the output translation template, which is commonly `./locales` in add-on packages.

If you wish to update the Zope 3 core message catalog template file, you simply run the extraction tool without specifying any options.

18.6 Step VI: Translating Message Strings

Now that we have a Message Catalog Template file, we can finally create a translation. Since we do not have existing message catalogs, you can simply copy the POT template file to the language you want to localize. In Unix you can just do the following from the `locales` directory:

```
cp messageboard.pot de/LC_MESSAGES/messageboard.po
```

Open `de/LC_MESSAGES/messageboard.po` in you favorite translation tool or a text editor. However, it is strongly recommended to use a gettext-specific translation

tool, since it will guarantee format integrity. Some of the choices include KBabel and the Vim/Emacs gettext modes.

KBabel seems to be the most advanced tool and develops to become a standard application for localization. It has many functions that make it easy for translators to do their job efficiently. My wife and I have translated many files using KBabel and it is a fantastic tool. It allows you, for example, to walk only through all untranslated or fuzzy strings and helps managing the message strings by providing message numbers and statistics.

After you are done with the translations, save the changes and you should be all set.

Great, we have a translation, but what happens if you develop new code and you need to update the template and catalog files? For creating the template you have to do nothing different, since a template can be created over and over from scratch. But this is not so easy with the actual catalogs, since you do not want to loose existing translations. The gettext utilities, that come with every Linux system, have a nice command line tool called `msgmerge` (for Windows you can just use Cygwin's version of the gettext packages). `msgemerge` merges all changes of the POT file into the message catalog, keeping all comments and existing translations intact and even marking changed translations as "fuzzy".

Here is how you can use the tool from the `locales` directory:

```
msgmerge -U de/LC_MESSAGES/messageboard.po ./messageboard.pot
```

18.7 Step VII: Compiling and Registering Message Catalogs

Before we can use our new translations, we need to compile the message catalogs into a more efficient binary format and then register the `locales` directory as a message catalog container.

To compile the catalogs, go to the directory and type:

```
msgfmt messageboard.po -o messageboard.mo
```

The `msgfmt` program is part of the gettext tools, which you must have installed (like for the `msgmerge` tool) to successfully execute the above command.

If you have troubles keeping the extensions `po` and `mo` in mind, here is a crib: The "p" of the ".po" extension stands for "people comprehensible" and the "m" in ".mo" for "machine comprehensible".

To register the `locales` directory as a translation container, open the main `configure.zcml` for the message board, and register the `i18n` namespace as follows in the `configure` tag:

```
1 xmlns:i18n="http://namespaces.zope.org/i18n"
```

Now you register the directory using

```
1 <i18n:registerTranslations directory="locales" />
```

The `i18n:registerTranslations` directive is smart enough to detect the directory structure and extract all the message catalogs for all the available languages.

An important note: During the last few steps it was quietly asserted that the filename of the message catalog *must* be the domain name! The `registerTranslations` directive uses the filename to determine the domain, which is completely in line with the gettext standard.

18.8 Step VIII: Trying the Translations

To test the translations, restart Zope 3. Different languages are best tested with Mozilla, since it allows you to quickly change the accepted languages of the browser itself. You can change the language in the preferences under Navigator → Languages. Put `German[de]` at the top of the list. The best view to test is the `Preview`, which you can reach with a URL similar to:
`http://localhost:8080/board/msg/@@details.html`

You should now see all the attribute names (such as `Title`, which became `Titel`) in German. You should also notice that the date is formatted in the German standard way using "day.month.year" and a 24-hour time.

Figure 18.1: The Message Details view in German

18.9 Step IX: Updating Translations on the Fly

While translating a package, it might be very cumbersome to restart Zope 3 just to update translations. For this reason, a process control called "Translation Domain Control" (`http://localhost:8080/++etc++process/@@TranslationDomain.html`) was created that allows you to update message catalogs at runtime without needing to restart the server. You should see the new `messageboard` domain for German (`de`).

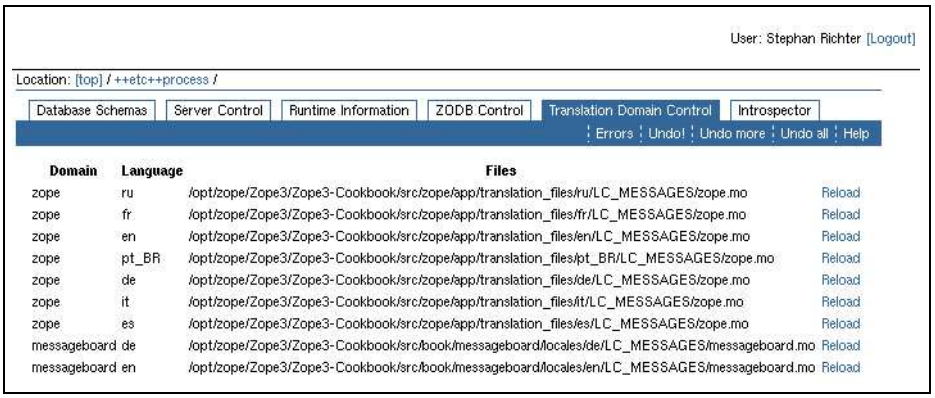


Figure 18.2: Translation Domain Control

Exercises

1. Complete the internationalization of all of the Page Templates.
2. Extract the new message strings (due to exercise 1) and merge them with the existing translations and update the English message catalog.

PART IV

Content Components – Advanced Techniques

Having a well-working basic message board is great, but it is certainly not blowing away anyone. In this section some more advanced APIs are presented.

Chapter 19: Events and Subscribers

Events et al are a very powerful idea. This chapter will explain how to write your own event subscribers by implementing a mail subscription feature for messages.

Chapter 20: Approval Workflow for Messages

This chapter will show how to integrate an editorial workflow for a content component.

Chapter 21: Providing Online Help Screens

Every good application should have Online Help screens, which is outlined in this chapter.

Chapter 22: Object to File System mapping using FTP as example

While there are standard hooks for content objects to be handled by FTP, it is often useful to write your own FTP handlers, so that the file-to-object conversion (and the other way around) seems more natural.

Chapter 23: Availability via XML-RPC

If you want to make XML-RPC calls on your content objects, you must write a view declaring the methods and define how their output is mapped to a type that XML-RPC understands.

Chapter 24: Developing new Skins

This chapter gives instructions on how to implement a new skin, so that sites can be developed that do not look like the Zope Management Interface, but still allows us to make use of all the auto-generation of forms.

CHAPTER 19

EVENTS AND SUBSCRIBERS

Difficulty

Contributor

Skills

- You should be comfortable with the topics covered in the "Content Components – The Basics" part.
- Feel comfortable with the Component Architecture.
- Be familiar with annotations. Read the appropriate chapters in this book, if necessary.

Problem/Task

Events are a powerful programming tool and are primary citizens in Zope 3. This chapter will concentrate on the subscription of existing events by implementing a mail subscription system for messages – whenever a message is modified, subscribers receive an E-mail about the change. This will also demonstrate again how annotations can be added to an object. In the last part of the chapter we will talk theoretically about triggering events.

Solution

There are two main components that need to be developed. The first is the mail subscription adapter for the message, which manages the subscription E-mails. The second component is the Event Subscriber , which listens for incoming events and starts the mailing process, if appropriate.

19.1 Step I: Mail Subscription Interface

We need to have an interface for managing the subscriptions for a particular message, i.e. add, delete and getting E-mail addresses. So add the following interface to the `interfaces` module:

```
class IMailSubscriptions(Interface):
    """This interface allows you to retrieve a list of E-mails for
    mailings. In our context these are messages."""

    def getSubscriptions():
        """Return a list of E-mails."""

    def addSubscriptions(emails):
        """Add a bunch of subscriptions; one would be okay too."""

    def removeSubscriptions(emails):
        """Remove a set of subscriptions."""
```

This code is simple enough, so that no further explanation is needed at this point.

19.2 Step II: Implementing the Mail Subscription Adapter

The implementation should be straightforward. The subscriptions are implemented as a simple tuple data structure, which are accessible via the annotation adapter. Note that the implementation makes no assumption about the type of annotation that is going to be used, i.e. we might have used the `AttributeAnnotations` out of pure convenience, but the data could just as well be stored in LDAP without having any effect on the `MailSubscriptions` implementation.

Since there is no need to create a new module, add the following code to the `message.py` file:

```
from zope.app.annotation.interfaces import IAnnotations
from book.messageboard.interfaces import IMailSubscriptions

SubscriberKey='http://www.zope.org/messageboard#1.0/MailSubscriptions/emails'


class MailSubscriptions:
    """Message Mail Subscriptions."""

    implements(IMailSubscriptions)
    __used_for__ = IMessage

    def __init__(self, context):
        self.context = self.__parent__ = context
        self._annotations = IAnnotations(context)
        if not self._annotations.get(SubscriberKey):
            self._annotations[SubscriberKey] = ()

    def getSubscriptions(self):
```

```
20          "See book.messageboard.interfaces.IMailSubscriptions"
21          return self._annotations[SubscriberKey]
22
23      def addSubscriptions(self, emails):
24          "See book.messageboard.interfaces.IMailSubscriptions"
25          subscribers = list(self._annotations[SubscriberKey])
26          for email in emails:
27              if email not in subscribers:
28                  subscribers.append(email.strip())
29          self._annotations[SubscriberKey] = tuple(subscribers)
30
31      def removeSubscriptions(self, emails):
32          "See book.messageboard.interfaces.IMailSubscriptions"
33          subscribers = list(self._annotations[SubscriberKey])
34          for email in emails:
35              if email in subscribers:
36                  subscribers.remove(email)
37          self._annotations[SubscriberKey] = tuple(subscribers)
```

- ▷ Line 4: This is the fully qualified subscriber annotation key that will uniquely identify this annotation data. Here a URL is used, but dotted names are also common.
- ▷ Line 11: While this declaration is not needed, it clearly signifies that this implementation is an adapter for `IMessage` objects.
- ▷ Line 14: Since this adapter will use annotations, it will be a trusted adapter, meaning that it will be a proxied object. All proxied objects must provide a location (at least through a `__parent__` attribute) so that permission declarations can be found. Otherwise only global permission settings would be available.
- ▷ Line 15: Here we are getting the `Annotations` adapter that will provide us with a mapping object in which we will store the annotations. Note that this statement says nothing about the type of annotation we are about to get.
- ▷ Line 16–17: Make sure an entry for our subscriber key exists. If not, create an empty one.
- ▷ Line 19–37: There is nothing interesting going on here. The only fact worth mentioning is the use of tuples instead of lists, which make the code a bit more complex, but tuples are not mutable, so that they are automatically saved in the ZODB, *if* we have `AttributeAnnotations`.

This is pretty much everything that is to the subscription part of this step. We can now register the new component via ZCML using the adapter directive:

```
1 <adapter
2     factory=".message.MailSubscriptions"
3     provides=".interfaces.IMailSubscriptions"
```

```
4     for=".interfaces.IMessage"
5     permission="book.messageboard.Add"
6     trusted="true" />
```

▷ Line 2–4: Like for the `ISized` adapter, we specify the necessary adapter registration information.

▷ Line 6: If an adapter is declared trusted then its context (the object being passed into the adapter constructor) will *not* be security proxied. This is necessary so that the annotations adapter can use the `__annotations__` attribute to store the annotations. If the adapter is not trusted and the context is security proxied, then a `ForbiddenAttribute` error will be raised whenever we try to access the annotations.

▷ Line 5: Once an adapter is trusted, the adapter itself is security proxied. Therefore we need to define a permission that is required to use the adapter.

19.3 Step III: Test the Adapter

The tests are as straightforward as the implementation. In the doc string of the `MailSubscriptions` class add the following documented testing code.

```
1  Verify the interface implementation
2
3  >>> from zope.interface.verify import verifyClass
4  >>> verifyClass(IMailSubscriptions, MailSubscriptions)
5  True
6
7  Create a subscription instance of a message
8
9  >>> msg = Message()
10 >>> sub = MailSubscriptions(msg)
11
12 Verify that we have initially no subscriptions and then add some.
13
14 >>> sub.getSubscriptions()
15 ()
16 >>> sub.addSubscriptions(('foo@bar.com',))
17 >>> sub.getSubscriptions()
18 ('foo@bar.com',)
19 >>> sub.addSubscriptions(('blah@bar.com',))
20 >>> sub.getSubscriptions()
21 ('foo@bar.com', 'blah@bar.com')
22 >>> sub.addSubscriptions(('doh@bar.com',))
23 >>> sub.getSubscriptions()
24 ('foo@bar.com', 'blah@bar.com', 'doh@bar.com')
25
26 Now let's also check that we can remove entries.
27
28 >>> sub.removeSubscriptions(('foo@bar.com',))
29 >>> sub.getSubscriptions()
```

```
30 ('blah@bar.com', 'doh@bar.com')
31
32 When we construct a new mail subscription adapter instance, the values
33 should still be there.
34
35 >>> sub1 = MailSubscriptions(msg)
36 >>> sub1.getSubscriptions()
37 ('blah@bar.com', 'doh@bar.com')
```

- ▷ Line 3–5: Do a very detailed analysis to ensure that the `MailSubscriptions` class implements the `IMailSubscriptions` interface.
- ▷ Line 7–10: In doc tests it helps very much if you emphasize how you setup your test case. Here we make that very explicit by creating a separate section and add some explanation to it.
- ▷ Line 12–24: Check that we can retrieve the list of subscribers and add new ones as well.
- ▷ Line 26–30: Make sure deleting subscriptions works as well.
- ▷ Line 32–37: When we create a new adapter using the same message, the subscriptions should still be available. This ensures that the data is not lost when the adapter is destroyed. An even stronger test would be that the persistence also works.

Note that there is no check for the case the annotation is not there. This is due to the fact that the `MailSubscriptions` constructor should make sure the annotation is available, even though this means to simply create an empty storage, so we have definitely covered this case in the implementation.

Since the adapter uses annotations, it requires some setup of the component architecture to run the tests. We already bring the services up for the tests, but now we also have to register an adapter to provide the annotations. Therefore we have to write a custom `setUp()` method and use it. The testing code in `tests/test_message.py` changes to:

```
1 from zope.interface import classImplements
2
3 from zope.app.annotation.attribute import AttributeAnnotations
4 from zope.app.interfaces.annotation import IAnnotations
5 from zope.app.interfaces.annotation import IAttributeAnnotatable
6 from zope.app.tests import placelesssetup
7 from zope.app.tests import ztapi
8
9 def setUp(test):
10     placelesssetup.setUp()
11     classImplements(Message, IAttributeAnnotatable)
12     ztapi.provideAdapter(IAttributeAnnotatable, IAnnotations,
13                          AttributeAnnotations)
```

```
14
15 def test_suite():
16     return unittest.TestSuite((
17         DocTestSuite('book.messageboard.message',
18                      setUp=setUp, tearDown=placelesssetup.tearDown),
19         unittest.makeSuite(Test),
20         ))
```

- ▷ Line 7: The `ztapi` module contains some very useful convenience functions to set up the component architecture for a test, such as view and adapter registration.
- ▷ Line 9: Note that the `setUp()` expects a `test` argument which is an instance of `DocTest`. You can use this object to provide global test variables.
- ▷ Line 11: We usually use ZCML to declare that `Message` implements `IAttributeAnnotatable`. Since ZCML is not executed for unit tests, we have to do it manually here.
- ▷ Line 12–13: Setup the adapter that allows us to look up an annotations adapter for any object claiming it is `IAttributeAnnotatable`.

You should now run the tests and ensure they pass.

19.4 Step IV: Providing a View for the Mail Subscription

The last piece we have to provide is a view to manage the subscriptions via the Web. The page template (`subscriptions.pt`) could look like this:

```
1  <html metal:use-macro="views/standard_macros/view">
2    <body>
3      <div metal:fill-slot="body" i18n:domain="messageboard">
4
5        <form action="changeSubscriptions.html" method="post">
6
7          <div class="row">
8              <div class="label"
9                   i18n:translate="">Current Subscriptions</div>
10             <div class="field">
11            <div tal:repeat="email view/subscriptions">
12                 <input type="checkbox" name="remails:list"
13                        value="" tal:attributes="value email">
14                 <div tal:replace="email">zope3@zope3.org</div>
15               </div>
16               <input type="submit" name="REMOVE" value="Remove"
17                    i18n:attributes="value remove-button">
18             </div>
19         </div>
20
21         <div class="row">
22             <div class="label" i18n:translate="">
23               Enter new Users (separate by 'Return')
24             </div>
```

```
25            <div class="field">
26           <textarea name="emails" cols="40" rows="10"></textarea>
27            </div>
28        </div>
29
30             <div class="row">
31               <div class="controls">
32                 <input type="submit" value="Refresh"
33                i18n:attributes="value refresh-button" />
34                 <input type="submit" name="ADD" value="Add"
35                     i18n:attributes="value add-button" />
36               </div>
37             </div>
38
39      </form>
40
41    </div>
42  </body>
43 </html>
```

▷ Line 7–19: The first part lists the existing subscriptions and let's you select them for removal.

▷ Line 20–38: The second part provides a textarea for adding new subscriptions. Each E-mail address should be separated by a newline (one E-mail per line).

The supporting View Python class then simply needs to provide a `subscriptions()` method (see line 11 above) and a form action. Place the following code into `browser/message.py`:

```
1  from book.messageboard.interfaces import IMailSubscriptions
2
3  class MailSubscriptions:
4
5      def subscriptions(self):
6          return IMailSubscriptions(self.context).getSubscriptions()
7
8      def change(self):
9          if 'ADD' in self.request:
10             emails = self.request['emails'].split('\n')
11             IMailSubscriptions(self.context).addSubscriptions(emails)
12         elif 'REMOVE' in self.request:
13             emails = self.request['remails']
14             if isinstance(emails, (str, unicode)):
15                 emails = [emails]
16             IMailSubscriptions(self.context).removeSubscriptions(emails)
17
18         self.request.response.redirect('./@@subscriptions.html')
```

▷ Line 9 & 12: We simply use the name of the submit button to decide which action the user intended.

The rest of the code should be pretty forward. The view can be registered as follows:

```
1  <pages
2      for="book.messageboard.interfaces.IMessage"
3      class=".message.MailSubscriptions"
4      permission="book.messageboard.Edit"
5      >
6    <page
7        name="subscriptions.html"
8        template="subscriptions.pt"
9        menu="zmi_views" title="Subscriptions"
10       />
11   <page
12       name="changeSubscriptions.html"
13       attribute="change"
14       />
15 </pages>
```

▷ Line 1: The `browser:pages` directive allows us to register several pages for an interface using the same view class and permission at once. This is particularly useful for views that provide a lot of functionality.

▷ Line 6–10: This page uses a template for creating the HTML.

▷ Line 11–14: This view on the other hand, uses an attribute of the view class. Usually methods on the view class do not return HTML but redirect the browser to another page.

▷ Line 9: Make sure the `Subscriptions` view becomes a tab for the `Message` object.

It is amazing how compact the `browser:pages` and `browser:page` directives make the registration. In the early development stages we did not have this directive and everything had to be registered via `browser:view`, which required a lot of repetitive boilerplate in the ZCML and Python code.

19.5 Step V: Message Mailer – Writing an Event Subscriber

Until now we have not even said one word about events. But this is about to change, since the next task is to implement the subscriber object. The generic event system is very simple: It consists of a list of subscribers and a `notify()` function. Subscribers can be subscribed to the event system by appending them to the list. To unsubscribe an object it must be removed from the list. Subscribers do not have to be any special type of objects; they merely have to be callable. The `notify()` function takes an object (the event) as a parameter; it then iterates though the list and calls each subscriber passing through the event.

This means that we have to implement a `__call__()` method as part of our message mailer API in order to make it a subscriber. The entire `MessageMailer` class should look like this (put it in the `message` module):

```
from zope.app import zapi
from zope.app.container.interfaces import IObjectAddedEvent
from zope.app.container.interfaces import IObjectRemovedEvent
from zope.app.event.interfaces import IObjectModifiedEvent
from zope.app.mail.interfaces import IMailDelivery

class MessageMailer:
    """Class to handle all outgoing mail."""

    def __call__(self, event):
        """Called by the event system."""
        if IMessage.providedBy(event.object):
            if IObjectAddedEvent.providedBy(event):
                self.handleAdded(event.object)
            elif IObjectModifiedEvent.providedBy(event):
                self.handleModified(event.object)
            elif IObjectRemovedEvent.providedBy(event):
                self.handleRemoved(event.object)

    def handleAdded(self, object):
        subject = 'Added: '+zapi.getName(object)
        emails = self.getAllSubscribers(object)
        body = object.body
        self.mail(emails, subject, body)

    def handleModified(self, object):
        subject = 'Modified: '+zapi.getName(object)
        emails = self.getAllSubscribers(object)
        body = object.body
        self.mail(emails, subject, body)

    def handleRemoved(self, object):
        subject = 'Removed: '+zapi.getName(object)
        emails = self.getAllSubscribers(object)
        body = subject
        self.mail(emails, subject, body)

    def getAllSubscribers(self, object):
        """Retrieves all email subscribers."""
        emails = ()
        msg = object
        while IMessage.providedBy(msg):
            emails += tuple(IMailSubscriptions(msg).getSubscriptions())
            msg = zapi.getParent(msg)
        return emails

    def mail(self, toaddrs, subject, body):
        """Mail out the Message Board change message."""
        if not toaddrs:
            return
        msg = 'Subject: %s\n\n\n%s' %(subject, body)
        mail_utility = zapi.getUtility(IMailDelivery, 'msgboard-delivery')
        mail_utility.send('mailer@messageboard.org' , toaddrs, msg)

```

```
55 mailer = MessageMailer()
```

- ▷ Line 2–4: We want our subscriber to handle add, edit and delete events. We import the interfaces of these events, so that we can differentiate among them.
- ▷ Line 10–18: This is the heart of the subscriber and this chapter. When an event occurs the `__call__()` method is called. First we need to check whether the event was caused by a change of an `IMessage` object; if so, let's check which event was triggered. Based on the event that occurred, a corresponding handler method is called.
- ▷ Line 20–36: These are the three handler methods that handle the various events. Note that the modified event handler should really generate a nice diff, instead of sending the entire message again.
- ▷ Line 38–45: This method retrieves all the subscriptions of the current message and all its ancestors. This way someone who subscribed to message `HelloEveryone` will also get e-mailed about all responses to `HelloEveryone`.
- ▷ Line 47–53: This method is a quick introduction to the Mail Delivery utility. Note how simple the `send()` method of the Mail Delivery utility is; it is the same API as for `smtplib`. The policy and configuration on how the mail is sent is fully configured via ZCML. See the configuration part later in this chapter.
- ▷ Line 60: We can only subscribe *callable* objects to the event system, so we need to instantiate the `MessageMailer` component.

Lastly, we need to register the message mailer component to the event service and setup the mail utility correctly. Go to your configuration file and register the following two namespaces in the `configure` element:

```
1 xmlns:mail="http://namespaces.zope.org/mail"
```

Next we setup the mail utility:

```
1 <mail:smtpMailer name="msgboard-smtp" hostname="localhost" port="25" />
2
3 <mail:queuedDelivery
4     name="msgboard-delivery"
5     permission="zope.SendMail"
6     queuePath="./mail-queue"
7     mailer="msgboard-smtp" />
```

- ▷ Line 1: Here we decided to send the mail via an SMTP server from `localhost` on the standard port 25. We could also have chosen to send the mail via the command line tool sendmail.

▷ Line 3–7: The Queued Mail Delivery utility does not send mails out directly but schedules them to be sent out independent of the current transaction. This has huge advantages, since the request does not have to wait until the mails are sent. However, this version of the Mail Utility requires a directory to store E-mail messages until they are sent. Here we specify the `mail-queue` directory inside the message board package. The value of the attribute `name` is used by the `MessageMailer` to retrieve the Queued Mail Delivery utility. Another Mail utility is the Direct Mail Delivery utility, which blocks the request until the mails are sent.

Now we register our message mailer object for the events we want to observe:

```
1 <subscriber
2     factory=".message.mailer"
3     for="zope.app.event.interfaces.IObjectModifiedEvent" />
4
5 <subscriber
6     factory=".message.mailer"
7     for="zope.app.container.interfaces.IObjectAddedEvent" />
8
9 <subscriber
10    factory=".message.mailer"
11    for="zope.app.container.interfaces.IObjectRemovedEvent" />
```

The `subscriber` directive adds a new subscriber (specified via the `factory` attribute) to the subscriber list. The `for` attribute specifies the interface the event must implement for this subscriber to be called. You might be wondering at this point why such strange attribute names were chosen. In the Zope application server, subscriptions are realized via adapters. So internally, we registered an adapter from `IObjectModifiedEvent` to `None`, for example.

Now you might think: "Oh let's try the new code!", but you should be careful. We should write some unit tests before testing the code for real.

19.6 Step VI: Testing the Message Mailer

So far we have not written any complicated tests in the previous chapters of the "Content Components – The Basics" part. This changes now. First of all, we have to bring up quite a bit more of the framework to do the tests. The `test_message.py` module's `setUp()` function needs to register the location adapters and the message mail subscription adapter. So it should look like that:

```
1 from zope.app.location.traversing import LocationPhysicallyLocatable
2 from zope.app.location.interfaces import ILocation
3 from zope.app.traversing.interfaces import IPhysicallyLocatable
4
5 from book.messageboard.interfaces import IMailSubscriptions
6 from book.messageboard.interfaces import IMessage
```

```
7  from book.messageboard.message import MailSubscriptions
8
9  def setUp():
10     ...
11     ztapi.provideAdapter(ILocation, IPhysicallyLocatable,
12                          LocationPhysicallyLocatable)
13     ztapi.provideAdapter(IMessage, IMailSubscriptions, MailSubscriptions)
```

▷ Line 1–3 & 11–12: This adapter allows us to use the API to access parents of objects or even the entire object path.

▷ Line 5–7 & 13: We simply register the mail subscription adapter that we just developed, so that the mailer can find the subscribers in the messages.

▷ Line 10: The three dots stand for the existing content of the function.

Now all the preparations are made and we can start writing the doctests. Let's look at the `getAllSubscribers()` method tests. We basically want to produce a message and add a reply to it. Both messages will have a subscriber. When the `getAllSubscribers()` method is called using the reply message, the subscribers for the original message and the reply should be returned. Here is the test code, which you should simply place in the `getAllSubscribers()` docstring:

```
1  Here a small demonstration of retrieving all subscribers.
2
3  >>> from zope.interface import directlyProvides
4  >>> from zope.app.traversing.interfaces import IContainmentRoot
5
6  Create a parent message as it would be located in the message
7  board. Also add a subscriber to the message.
8
9  >>> msg1 = Message()
10 >>> directlyProvides(msg1, IContainmentRoot)
11 >>> msg1.__name__ = 'msg1'
12 >>> msg1.__parent__ = None
13 >>> msg1_sub = MailSubscriptions(msg1)
14 >>> msg1_sub.context.__annotations__[SubscriberKey] = ('foo@bar.com',)
15
16 Create a reply to the first message and also give it a subscriber.
17
18 >>> msg2 = Message()
19 >>> msg2_sub = MailSubscriptions(msg2)
20 >>> msg2_sub.context.__annotations__[SubscriberKey] = ('blah@bar.com',)
21 >>> msg1['msg2'] = msg2
22
23 When asking for all subscriptions of message 2, we should get the
24 subscriber from message 1 as well.
25
26 >>> mailer.getAllSubscribers(msg2)
27 ('blah@bar.com', 'foo@bar.com')
```

▷ Line 3–4: Import some of the general functions and interfaces we are going to use for the test.

▷ Line 6–14: Here the first message is created. Note how the message must be a `IContainmentRoot` (line 10). This signalizes the traversal lookup to stop looking any further once this message is found. Using the mail subscription adapter (line 13–14), we now register a subscriber for the message.

▷ Line 16–21: Here we create the reply to the first message. The parent and name of the second message will be automatically added during the `__setitem__` call.

▷ Line 23–27: The mailer should now be able to retrieve both subscriptions. If the test passes, it does.

Finally we test the `__call__()` method directly, which is the heart of this object and the only *public* method. For the notification to work properly, we have to create and register an `IMailDelivery` utility with the name "msgboard-delivery". Since we do not want to actually send out mail during a test, it is wise to write a stub implementation of the utility. Therefore, start your doctests for the `notify()` method by adding the following mail delivery implementation to the docstring of the method:

```
1  >>> mail_result = []
2
3  >>> from zope.interface import implements
4  >>> from zope.app.mail.interfaces import IMailDelivery
5
6  >>> class MailDeliveryStub(object):
7  ...     implements(IMailDelivery)
8  ...
9  ...     def send(self, fromaddr, toaddrs, message):
10 ...         mail_result.append((fromaddr, toaddrs, message))
11
12 >>> from zope.app.tests import ztapi
13 >>> ztapi.provideUtility(IMailDelivery, MailDeliveryStub(),
14 ...                      name='msgboard-delivery')
```

▷ Line 1: The mail requests are stored in this *global* variable, so that we can make test assertions about the supposedly sent mail.

▷ Line 6–10: Luckily the Mail utility requires only the `send()` method to be implemented and there we simply store the data.

▷ 12–14: Using the `ztapi` API, we can quickly register the utility. Be careful that you get the name right, otherwise the test will not work.

So far so good. Like for the previous test, we now have to create a message and add a subscriber.

```
1  Create a message.
2
3  >>> from zope.interface import directlyProvides
```

```
>>> from zope.app.traversing.interfaces import IContainmentRoot

>>> msg = Message()
>>> directlyProvides(msg, IContainmentRoot)
>>> msg.__name__ = 'msg'
>>> msg.__parent__ = None
>>> msg.title = 'Hello'
>>> msg.body = 'Hello World!'

Add a subscription to message.

>>> msg_sub = MailSubscriptions(msg)
>>> msg_sub.context.__annotations__[SubscriberKey] = ('foo@bar.com',)
```

This is equivalent to what we did before, so nothing new here. Finally, we create an modification event using the message and send it to the `notify()` method. We then problem the global `mail_result` variable for the correct functioning of the method.

```
Now, create an event and send it to the message mailer object.

>>> from zope.app.event.objectevent import ObjectModifiedEvent
>>> event = ObjectModifiedEvent(msg)
>>> mailer(event)

>>> from pprint import pprint
>>> pprint(mail_result)
[('mailer@messageboard.org',
  ('foo@bar.com',),
  'Subject: Modified: msg\n\n\nHello World!')]
```

- ▷ Line 3–4: In this particular test, we use the object modification event. Any `IObjectEvent` can be initiated by passing the affected object as argument to the constructor of the event.
- ▷ Line 5: Here we notify the mailer that an object has been modified. Note that the `mailer` is an instance of the `MessageMailer` class and is initialized at the end of the module.
- ▷ Line 7–11: The pretty print (`pprint`) module comes in very handy when outputting complex data structures.

We are finally done now. You should run the tests to verify your implementation and then head over to the next section to see how we can give this code a real swirl.

19.7 Step VII: Using the new Mail Subscription

First of all, we have to restart Zope and make sure in boots up properly. Then you can go to the management interface and view a particular message. You might notice now that you have a new tab called `Subscriptions`, so click on it.

In the `Subscriptions` view, you will see a text area in which you can enter subscription E-mail addresses, which will receive E-mails when the message or any children are changed. When adding a test E-mail address, make sure this E-mail address exists and is your own, so you can verify its arrival. Click on the `Add` submit button to add the E-mail to the subscriber list. Once the screen returns, you will see this E-mail appear under "Current Subscriptions" with a checkbox before it, so you can delete it later, if you wish.

Next, switch to the `Edit` view and modify the `MessageBody` a bit and submit the change. You should notice that the screen returns almost immediately, but that your mail has not necessarily arrived yet. This is thanks to the Queued Mail Delivery Utility, which sends the mails on a separate thread. However, depending on the speed of your E-mail server, a few moments later you should receive an appropriate E-mail.

19.8 Step VIII: The Theory

While this chapter demonstrates a classical use of events from an application developer point of view, it is not quite the whole story. So far we have discovered the use of the basic event system.

We did not explain how Zope uses this system. As mentioned before, the `subscriber` directive does not append the message mailer instance to the subscription list directly, as one may expect. Instead, it registers the message mailer as a "subscription adapter" that adapts the an event providing some event interface, i.e. `IObjectModifiedEvent`, to `None`, since it explicitly does not provide any special interface. The difference between regular and subscription adapters is that one can register several subscription adapters having the same required and provided provided interfaces. When requested, all matching adapters are returned. This allows us to have multiple subscribers for an event.

The Zope application server adds a special dispatch subscriber (`zope.app.event.dispatching`) that forwards the notification to all adapter-based subscriptions. In the following diagram you can see how an event flows through the various parts of the system to the subscriber that will make use of the event. The example is based on the code developed in this chapter.

A special kind of subscribers are event channels, which change an event and redistribute it or serve as event filters. You could think of them as middle men. We could have written this chapter using event channels by implementing a subscriber that forwards an event only, if the object provides `IMessage`. An implementation could look as follows:

```
1 def filterEvents(event):
2     if IMessage.providedBy(event.object):
3         zope.event.notify(event.object, event)
```

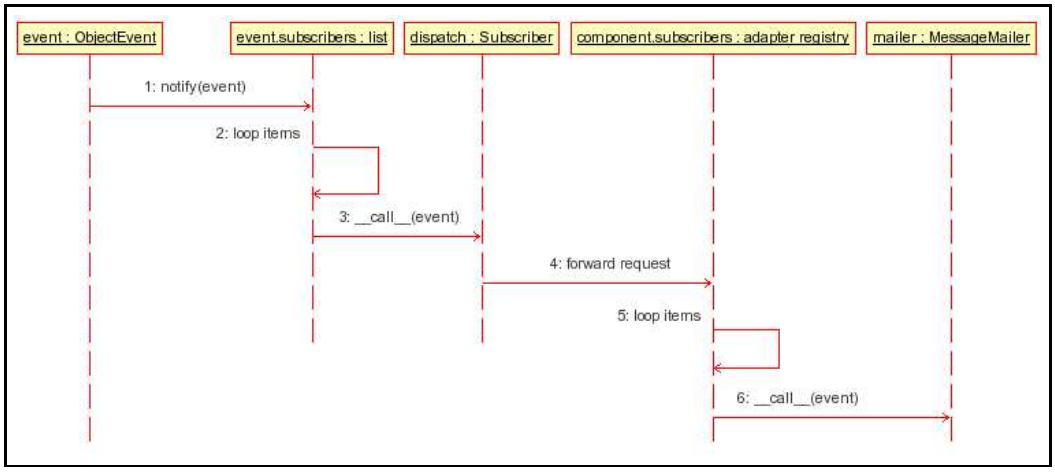


Figure 19.1: Demonstration of an event being distributed to its subscribers.

The actual mailer would then be a multi-adapter that adapts from both the message and the event:

```
class MessageMailer:

    __call__(self, message, event):
        ...
```

Multi-subscriptions-adapters are registered in ZCML as follows:

```
<subscriber
  factory = ".message.mailer"
  for = ".interface.IMessage
         zope.app.event.interface.IObjectEvent" />
```

The modified sequence diagram can be seen below in figure 19.8.

A final tip: Sometimes events are hard to debug. In these cases it is extremely helpful to write a small subscriber that somehow logs all events. In the simplest case this can be a one-line function that prints the string representation of the event in console. To subscribe a subscriber to all events, simply specify `for="*"` in the `zope:subscriber` directive.

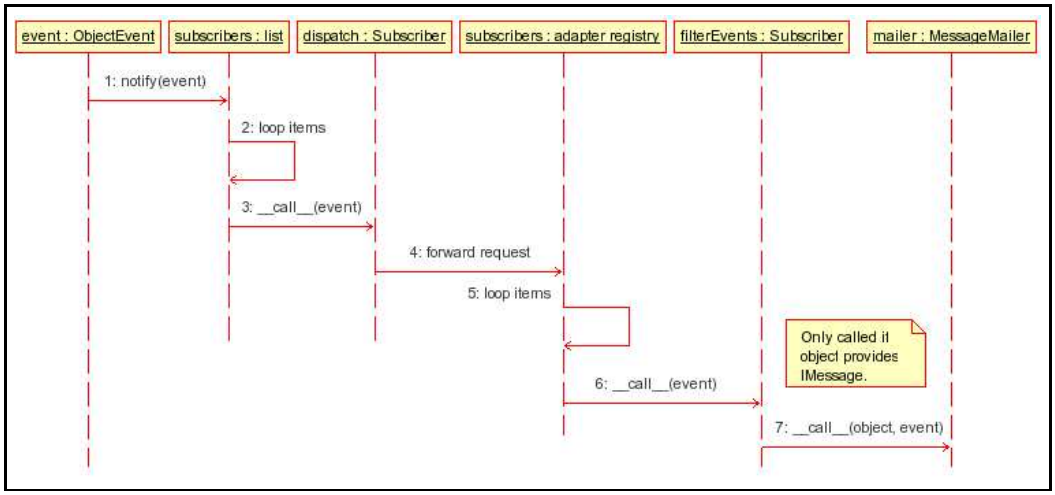


Figure 19.2: Modification of the even publication using an event channel.

Exercises

1. Finish the outlined implementation of the event channel above and rewrite the message mailer to be a multi-adapter.

2. Implement a subscriber that subscribes to all events and prints a line to the console for received event. Then extend this subscriber to use the common logging mechanism.

CHAPTER 20

APPROVAL WORKFLOW FOR MESSAGES

Difficulty

Contributor

Skills

- A good understanding of the Component Architecture and ZCML is required.
- Familiarity with the `messageboard` package is necessary, so that you can easily follow the new extensions.
- Some familiarity with workflows and various solutions is desired. Optional.

Problem/Task

Workflows are important in any company. Therefore it is not surprising that software-based workflows became a primary requirement for many computer systems, especially for content management systems (CMS). This chapter will add publication workflow to the `messageboard`.

Solution

While this chapter does not deal with every aspect of the `zope.app.workflow` package – for example it does not explain how to create Process Definitions – it demonstrates the most common use cases by integrating a workflow in an existing content object package. And the realization is amazingly simple. Behind the simple

frontend, however, there is a maze of interfaces, their implementation and their presentation. The goal of the last section of the chapter is then to explain the framework from an architectural point of view.

20.1 Step I: Making your Message Workflow aware

In order to make a content object workflow-aware, you simply have to tell the system that it can store workflow data. The simplest way is to add the following interface declaration to the `Message` content directive in the main configuration file:

```
<implements interface=
    "zope.app.workflow.interfaces.IProcessInstanceContainerAdaptable"/>
```

Appropriate adapters for storing workflow data are already defined for objects also implementing `IAnnotable`. Our message object does this already by implementing `IAttributeAnnotable` as you can see in the same content directive.

Now the object can contain workflows and when you restart your browser, you should notice that `Message` instances now also have a "Workflow" tab, which is still totally empty.

20.2 Step II: Create a Workflow and its Supporting Components via the Browser

Next we have to create the workflow components themselves. For this first attempt we are going to create all the components by hand, since this process provides some enlightenment on how the entire workflow mechanism works.

After you started Zope 3, go to the folder you want to add (or have already) your messageboard. Go to the Site Manager by clicking on "Manage Site"; if the Folder is not yet a site, click on "Make a site". Now click on the "Tools" tab.

If you you just created the site, you now have to create a "Local Utility Service". This can be accomplished by clicking on the "Service Tool" link. Click on the "Add" button, select the "Utility Service" on the next screen, enter a name (like "utilities" for example) and confirm all this by pressing "Add". Now you have a fully configured and activated local utility service. Go back to the tools overview screen now.

The next step is to define an actual workflow in terms of a process, which contains states and transitions. Therefore, click on the "Workflows" link. Next add a workflow by click on the "Add" button. For this workflow we want a "Stateful Process Definition" (which is likely to be you only choice) and name it "publish-message". Clicking on the "Add" button will create the workflow and activate it.

Since the stateful process defintion component supports a nice XML Import and Export filter, it is best to define the process in XML. For later reference, we are going

to store the workflow XML in a file and make it part of our product. Therefore, open a new file called `workflow.xml` in the `messageboard` package and add the following process definitions:

```
<?xml version="1.0"?>
<workflow type="StatefulWorkflow" title="Message Publication Review">
  <schema name=""/>
  <states>
    <state name="INITIAL" title="initial" />
    <state name="private" title="Private" />
    <state name="pending" title="Pending Publication" />
    <state name="published" title="Public" />
  </states>
  <transitions>

    <transition
        sourceState="published"
        destinationState="private"
        name="published_private"
        title="Unpublish Message"
        permission="book.messageboard.PublishContent"
        triggerMode="Manual" />

    <transition
        sourceState="private"
        destinationState="pending"
        name="private_pending"
        title="Submit Message"
        permission="book.messageboard.Edit"
        triggerMode="Manual" />

    <transition
        sourceState="INITIAL"
        destinationState="private"
        name="initial_private"
        title="Make Private"
        triggerMode="Automatic" />

    <transition
        sourceState="pending"
        destinationState="published"
        name="pending_published"
        title="Publish Message"
        permission="book.messageboard.PublishContent"
        triggerMode="Manual" />

    <transition
        sourceState="pending"
        destinationState="private"
        name="pending_private"
        title="Retract Message"
        permission="book.messageboard.Edit"
        triggerMode="Manual" />

    <transition
        sourceState="pending"
        destinationState="private"
        name="pending_private_reject"
```

```
55          title="Reject Message"
56          permission="book.messageboard.PublishContent"
57          triggerMode="Manual" />
58
59    </transitions>
60
61  </workflow>
```

- ▷ Line 2: Define the workflow to be a stateful workflow, the only type that is currently implemented. The "title" is the string under which the workflow will be known.
- ▷ Line 3: We do not have a particular data schema, so let's skip that. These schemas are used to allow the developer to add additional workflow-relevant data (object-specific) to the workflow instances.
- ▷ Line 4–9: Define the states a Message can be into. The title, again, serves as a human readable presentation of the state.
- ▷ Line 10–59: This is a list of all possible transitions the object can undergo. I think the attributes of the transition directive are self explanatory and do not need any further explanation.

Once you saved the XML file, click on the newly created workflow (in the "Workflows" tool overview) and then on the "Import/Export" tab. Copy the XML from the file and paste it into the textarea of the screen. Then press the "Import" button. The same screen will return with a message saying "Import was successfull!". You will also see the XML (probably differently formatted) at the botton of the screen. If you now click on `ManageStates`, you should see the fours states you just added via the XML import. The same is true for the `ManageTransitions` view.

You might have already noticed that the workflow requires a new permission named "book.messageboard.PublishContent" to be defined. Therefore go to the `messageboard`'s configuration file and add the permission:

```
<permission
    id="book.messageboard.PublishContent"
    title="Publish Message"
    description="Publish Message."/>
```

In the `security.zcml` configuration file grant the "Editor" the permission to publish content.

```
<grant
    permission="book.messageboard.PublishContent"
    role="book.messageboard.Editor"/>
```

Now restart Zope 3. That should be everything that's to it! Now let's see whether everything works.

20.3 Step III: Assigning the Workflow

Now that our message is workflow-aware and a workflow has been created, we have to assign the workflow to `IMessage` objects. This is done via the "Content Workflow Manager", which maps workflows to content objects.

Go to the site's "tools" Site Management Folder. To do that go to the site's overview and select the "Software" tab. You can now enter the "tools" folder. Once there, add a "Content Workflow Manager" named "ContentWorkflows". When completed, you are automatically forwarded to the "Registration" view, since the manager is just another utility. Click on the "Register" button, register the utility as "ContentWorkflows" and press the "Add" button. You have now successfully registered and activated the utility.

The next step is to declare the workflow to interface mapping. To do so, go to the "Content/Process Registry" tab of the workflow manager. On this page you should now see a list of interfaces (many of them) and a list of process definition names, which only contains one entry, the name of our previously created workflow. Select the `book.messageboard.interface.IMessage` interface and the "publish-message" and click on "Add Mappings". The previous page should return, but this time with an entry below "Available Mappings".

But how does the workflow gets appended to a message object? The content workflow manager is a subscriber to `IObjectCreated` events. If the created object implements an interface for which we have a workflow, then a process instance of this workflow is added to the object as an annotation. Note that one can assign many different workflows to an object. The workflow manager is subscribed as soon as you make it active as utility, which we already did when we registered it.

20.4 Step IV: Testing the Workflow

The workflow will only work with new Messages, of course. So, in the folder you created the workflow components, create a new Message Board and add a new Message to it. If you now click on the `Workflows` tab you will see that it is not empty anymore. In the selection box you can see all available workflows; currently there should be only one called "Message Publication Review" (remember the workflow title in the XML?). You can choose it.

Below the selection box you can see the current status of the Message, which is private at this point; remember, the transition from `initial` to `private` is automatic based on our workflow definition. In the last entry you now see the possible transitions you can execute from this state. Currently we can only "Submit

Message" to submit the message for review. So select this transition and click "Make Transition".

The status will switch to "Pending Publication" (pending) and now you have three transition choices. You might have noticed already that this workflow is not very safe or useful, since every Editor (and only editors) can cause a transitions. See exercise 1 to solve this problem. Feel free to play with the transitions some more.

20.5 Step V: Writing a nice "Review Messages" View for Message Boards

Now that we have the basic workflow working, we should look at an example on how we can make use of the workflow mechanism. So the first task will be to provide the Editor with a nice overview over all pending messages from the message board object.

So we basically need a view class that recursively walks through the tree and and picks out the pending messages. To do this, it is extremely helpful to write a convenience function that simply checks whether a message has a certain status. The implementation could be something line the following, which we simply place into `browser/messageboard.py`:

```
1  from zope.app import zapi
2  from zope.app.workflow.interfaces import IProcessInstanceContainer
3
4  def hasMessageStatus(msg, status, workflow='publish-message'):
5      """Check whether a particular message matches a given status"""
6      adapter = IProcessInstanceContainer(msg)
7      if adapter:
8          # No workflow is defined, so the message is always shown.
9          if not adapter.keys():
10             return True
11         for item in adapter.values():
12             if item.processDefinitionName != workflow:
13                 continue
14             if item.status == status:
15                 return True
16
17     return False
```

- ▷ Line 2 & 6: The returned adapter will provide us access to the message's workflows (process instances). in our case we only expect to find one workflow.
- ▷ Line 8–10: This is some backward compatibility for the messages that were created before we added the workflow feature.
- ▷ Line 11–15: Look through all the workflows (process instances) and try to find the one we are looking for. If the status matches the state we are checking for, then we can return a positive result. If not, we will eventually return `False` (Line 16).

Next we are going to implement the view class, which will provide the method `getPendingMessagesInfo()` which will return a list of information structures for pending messages, where each info contains the title and the URL to the workflow view of the message. Place the following view in `brower/messageboard.py`:

```
1  from book.messageboard.interfaces import IMessage
2
3  class ReviewMessages:
4      """Workflow: Review all pending messages"""
5
6      def getPendingMessages(self, pmsg):
7          """Get all pending messages recursively."""
8          msgs = []
9          for name, msg in pmsg.items():
10             if IMessage.providedBy(msg):
11                 if hasMessageStatus(msg, 'pending'):
12                     msgs.append(msg)
13                 msgs += self.getPendingMessages(msg)
14         return msgs
15
16     def getPendingMessagesInfo(self):
17         """Get all the display info for pending messages"""
18         msg_infos = []
19         for msg in self.getPendingMessages(self.context):
20             info = {}
21             info['title'] = msg.title
22             info['url'] = zapi.getView(
23                 msg, 'absolute_url', self.request)() + '/@@workflows.html'
24             msg_infos.append(info)
25         return msg_infos
```

▷ Line 6–14: This is the actual recursive method that searches for all the pending messages.

- Line 8: This will be the resulting flat list of pending messages.
- Line 10: Since we can find replies (messages) and attachments (files) in a message, we have to make sure that we deal with an `IMessage` object.
- Line 11–12: If the message is `pending`, then add it to the list of pending messages.
- Line 13: Whatever message it is, we definitely want to look at its replies to see whether there are pending messages lying around.

▷ Line 16–25: This method creates a list of infos about the messages using the list of pending messages (line 20). This is actually the method that will be called from the page template.

Next we create the template named `review.pt` that will display the pending messages:

```
1  <html metal:use-macro="views/standard_macros/view">
2    <body>
3      <div metal:fill-slot="body" i18n:domain="messageboard">
4
5        <h1 i18n:translate="">Pending Messages</h1>
6
7        <div class="row" tal:repeat="msg view/getPendingMessagesInfo">
8          <div class="field">
9            <a href="" tal:attributes="href msg/url"
10               tal:content="msg/title" />
11         </div>
12       </div>
13
14     </div>
15   </body>
16 </html>
```

▷ Line 7–12: Iterate over all message entries and create links to each pending message, displaying its title.

Finally, we just have to register the new view using simply:

```
1  <page
2      name="review.html"
3      for="book.messageboard.interfaces.IMessageBoard"
4      class=".messageboard.ReviewMessages"
5      permission="book.messageboard.PublishContent"
6      template="review.pt"
7      menu="zmi_views" title="Review Messages"/>
```

Now restart your Zope 3 server and enjoy the new view. You could do much more with this view, but this should give you an idea of the framework's functionality.

20.6 Step VI: Adjusting the Message Thread

Okay, now we have a working workflow, a way for the message writer to request publication and a view for the editor to approve or reject messages. But the workflow does not impinge on the interaction of the user with the message board at all yet. Therefore, let's modify the message thread view to only show published messages.

The change requires only two more lines in `browser/thread.py`. First import the `hasMessageStatus()` function.

```
1  from messageboard import hasMessageStatus
```

Second, extend the condition that checks that an object implements `IMessage` to also make sure it is published.

```
1  if IMessage.providedBy(child) and \
2         hasMessageStatus(child, 'published'):
```

And that's it! Restart Zope and check that it works!

20.7 Step VII: Automation of Workflow and Friends creation

Now that we have our workflow support completed, we should direct our attention to one last quirk. Remember when we created the workflow by hand in several steps. You certainly do not want to require your users to add all these objects by hand. It would be neat, if the workflow code would be added after the message board was created and added. And this is actually not hard to do. We only have to create a custom template and an add-view and insert it in the `browser:addform` directive.

So let's start with the template, which should provide an option for the user to choose whether the workflow objects should be generated or not. Create a new file called `messageboard_add.pt` and insert the following content:

```
<html metal:use-macro="views/standard_macros/page">
  <body>
    <div metal:fill-slot="body" i18n:domain="messageboard">

      <div metal:use-macro="views/form_macros/addform">

        <div metal:fill-slot="extra_bottom" class="row">
          <div class="field">
            <h3><input type="checkbox" name="workflow:int"
                    value="1" checked=""/>
              <span i18n:translate="">Create Workflow</span>
            </h3>
            <span i18n:translate="">Without the workflow you will
              not be able to review messages before they are
              published. Note that you can always modify the
              messageboard workflow later to make all transitions
              automatically.</span>
          </div>
        </div>

      </div>

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

Nothing surprising; if we find the `workflow` attribute in the request, we now the option was set. Next we write the custom create and add view for the messageboard, which I simply placed into `browser/messageboard.py`:

```
import os
from zope.proxy import removeAllProxies

from zope.app.registration.interfaces import ActiveStatus
from zope.app.site.interfaces import ISite
from zope.app.site.service import SiteManager, ServiceRegistration
from zope.app.utility.utility import LocalUtilityService, UtilityRegistration
from zope.app.workflow.interfaces import IProcessDefinitionImportHandler
from zope.app.workflow.stateful.contentworkflow import ContentWorkflowsManager
from zope.app.workflow.stateful.definition import StatefulProcessDefinition
from zope.app.workflow.stateful.interfaces import IContentWorkflowsManager
```

```
12 from zope.app.workflow.stateful.interfaces import IStatefulProcessDefinition
13
14 import book.messageboard
15
16
17 class AddMessageBoard(object):
18     """Add a message board."""
19
20     def createAndAdd(self, data):
21         content = super(AddMessageBoard, self).createAndAdd(data)
22
23         if self.request.get('workflow'):
24             folder = removeAllProxies(zapi.getParent(content))
25             if not ISite.providedBy(folder):
26                 sm = SiteManager(folder)
27                 folder.setSiteManager(sm)
28             default = zapi.traverse(folder.getSiteManager(), 'default')
29
30             # Create Local Utility Service
31             default['Utilities'] = LocalUtilityService()
32             rm = default.getRegistrationManager()
33             registration = ServiceRegistration(zapi.servicenames.Utilities,
34                                                'Utilities', rm)
35             key = rm.addRegistration(registration)
36             zapi.traverse(rm, key).status = ActiveStatus
37
38             # Create the process definition
39             default['publish-message'] = StatefulProcessDefinition()
40             pd_path = zapi.getPath(default['publish-message'])
41             registration = UtilityRegistration(
42                 'publish-message', IStatefulProcessDefinition, pd_path)
43             pd_id = rm.addRegistration(registration)
44             zapi.traverse(rm, pd_id).status = ActiveStatus
45
46             import_util = IProcessDefinitionImportHandler(
47                 default['publish-message'])
48
49             xml = os.path.join(
50                 os.path.dirname(book.messageboard.__file__), 'workflow.xml')
51
52             import_util.doImport(open(xml, mode='r').read())
53
54             # Create Content Workflows Manager
55             default['ContentWorkflows'] = ContentWorkflowsManager()
56             cm_path = zapi.getPath(default['ContentWorkflows'])
57             registration = UtilityRegistration(
58                 'wfcontentmgr', IContentWorkflowsManager, cm_path)
59             cm_id = rm.addRegistration(registration)
60             zapi.traverse(rm, cm_id).status = ActiveStatus
61
62             contentmgr = default['ContentWorkflows']
63             contentmgr.register(IMessage, 'publish-message')
64
65         return content
```

- Line 1–14: A huge amount of imports, so that all components are available. I think this alone shows what a mess simple configuration objects and ZCML usually save us from.
- Line 20–21: The `createAndAdd` method is the only one we have to override and extend. The good part is that the original method returns the added message board instance itself, so that we store it and make use of it. After this line, the message board is already created and added.
- Line 23: If the user wants us to autogenerate the workflow objects, then let's do it.
- Line 24: Grab the folder that contains the message board.
- Line 25–27: Make sure that the folder is a site. If not make it one.
- Line 28: Now we just get the `default` site management folder, into which we will place all the local component.
- Line 30–36: Create a new local utility service, so that we can register our local utilities we are about to create. Note that both the "Content Workflow Manager" and the "Stateful Process Definition" are local utilites.
- Line 38–44: Add the a new process definition and register it to be usable (active).
- Line 46–52: Here comes the tricky part. We have to create the workflow states and transitions from our saved `workflow.xml` file. But where to get the directory from? The easiest way is to import the package itself, get the path, then truncate the `__init__.py` part and we should be left with the directory path. You then simply add the workflow XML filename at the end and open it for import. The reason we want to use the `os` module everywhere is that we want to keep Zope packages platform-independent.
- Line 54–63: Create the content workflows manager, which gets notified when `IObjectCreatedEvent` events occurr, so it can add process instances to it. On line 63 we tell the system that the "publish-message" workflow (created above) should be used only for `IMessage` components.

Now we need to register the add view class and template with the addform in ZCML. The `addform` directive for the message board therefore becomes:

```
1 <addform
2     label="Add Message Board"
3     name="AddMessageBoard.html"
4     template="messageboard_add.pt"
5     class=".messageboard.AddMessageBoard"
```

```
6       schema="book.messageboard.interfaces.IMessageBoard"
7       content_factory="book.messageboard.messageboard.MessageBoard"
8       fields="description"
9       permission="zope.ManageContent"
10      />
```

▷ Line 3–4: See how easy it is to incorporate custom templates and classes for an add form (the same is true for edit forms).

After restarting Zope, you should be able to enjoy the changes. Create a new Folder and in it a new Message Board. You should now see the new option and after the message board was successfully created, the workflow components should be available in the parent folder.

20.8 The Theory

Now that we have completed the practical part of the chapter, we should look a bit more carefully at the framework supporting all this functionality. The framework was designed to support any type of generic workflow implementation. The Zope community itself has produced two, the "activity" and "entity" model.

Activity-based workflows implement workflow in a transition-centric fashion, where an object is moved in a graph of workflow states and transitions outside of its physical hierarchy. This type of model was developed by the Workflow Management Coalition (WfMC) and is implemented in the Zope 2 OpenFlow/CMFFlow product. The advantage of this model is a high degree of flexibility and scalability, which is well established thanks to the WfMC.

Entity-based workflows, on the other hand, store the current workflow state of the object as meta data in the object itself, so that no real workflow graph exists, but is only defined by a set of states and transitions. This model was implemented by `DCWorkflow` in Zope 2 and is known as "stateful" in Zope 3. One of its advantages is simplicity of implementation and a flatter learning curve. It is the workflow type we used in this chapter.

Some terms:

- Process Definition: This component defines in what states a content object can be and what the possible transitions between them are. It is basically a blue print of the actual workflow.
- Process Instance: If the Process Definition is the blueprint, then the Process Instance is the workflow itself; it is the realization of the Process Definition, which is used to actually manage the workflow for *one* particular object, i.e.

there is one Process Instance per workflow per content component instance. Note that one object can have several workflows associated with itself.

- Process Instance Container: This object is used to store actual Process Instances and is usually the component that is tagged to an object via an annotation.
- Content Workflows Manager (stateful): This utility is responsible to add the correct workflows to a content object upon creation.

One of the powerful features of the "stateful" workflow implementation is that every process instance can have workflow-relevant data associated with it. The specifics of this data are specified via a schema in the process definition. When an instance of a process is appended to an object, placeholders for this data are created as well. The workflow-relevant data can be useful for transition conditions, comments and the like.

Exercises

1. The current workflow is not very secure. Any message board Editor can cause all transitions. Therefore create a different permission for the "Submit Message" (private to pending) and the "Retract Message" (pending to private) transition and assign it to the Message Board user. Make sure that now users can only cause these two transitions and editors still can cause them all.

2. The `ReviewMessages` view is in some respects pretty boring and not very user-friendly. It would be nice to be able to mass-approve messages or reject them, in case of spamming. Extend the `ReviewMessages` to support this feature.

CHAPTER 21

PROVIDING ONLINE HELP SCREENS

Difficulty

Newcomer

Skills

- While this chapter has almost no direct prerequisites, the developer should still be familiar with at least the first two chapters of this section.
- Some ZCML knowledge is of advantage.

Problem/Task

Offering help to the user at any point of a GUI is an important feature for all applications. Our message board package does a really bad job with providing help up to this point. This chapter will change that by using Zope 3's online help package. This has not much to do with Python programming, but is part of the development process.

Solution

This should be a very quick chapter, since there are only two tasks. First you need to write the actual help screens (can be either pain text, STX, ReST or HTML) and then you simply register them. So let's dive right into it. Since the help will be for browser views, I prefer to place the help files in a `help` directory inside `messageboard/browser`.

First create a file called `package_intro.rst` and enter the following content:

```
==========================
Message Board Demo Package
==========================

This package demos various features of the Zope 3 Framework. If you
have questions or concerns, please let me know.
```

Then a file called `board_review.rst` containing

```
This view lists all messages in the board that are pending for
publication. Each listed method is a link that brings you to the
message's "Workflow" view where you can initiate a transition.
```

Finally add `msg_edit.rst` with the following text:

```
This screen allows you to edit the data (i.e. the subject and body) of
the Message object.

title - A one line unicode text string that briefly describes the
        purpose/subject of the message.

body - A multiple line unicode text string that is the actual content of
       the message. It is accepting HTML, but restricts the user to a
       couple of selected tags. Feel free to type anything you wish.
```

Notice how I do not have titles for the text itself. We will define titles in the configuration, which is displayed as header on the Web site, so that there is no need for another title.

All that's left to do is to register the new help screens. Help Topics can be organized in a hierarchical manner. In order to keep all of the message board package screens together in one sub-tree, we make the `package_info.rst` help topic the parent of all the other help screens. Open your configuration file (`messageboard/browser/configure.zcml`). Then we need to add the `help` namespace in the `zope:configure` element using the following declaration:

```
xmlns:help="http://namespaces.zope.org/help"
```

Now you can add the following directives:

```
<help:register
    id="messageboard"
    title="Message Board Help"
    parent="ui"
    for="book.messageboard.interfaces.IMessageBoard"
    doc_path="./help/package_intro.rst"/>

<help:register
    id="board.review"
    title="Publication Review"
    parent="ui/messageboard"
    for="book.messageboard.interfaces.IMessageBoard"
    view="review.html"
    doc_path="./help/board_review.rst"/>

<help:register
    id="message.edit"
```

```
18      title="Change Message"
19      parent="ui/messageboard"
20      for="book.messageboard.interfaces.IMessage"
21      view="edit.html"
22      doc_path="./help/msg_edit.rst"/>
```

- Line 2: This is the id of the Help Topic as it will be available in the URL.
- Line 3: The title of the Help Topic that is displayed above the topics content.
- Line 4: The path of the parent help topic. The `ui` Help Topic comes by default with Zope 3, and you should attach all your "screen help" to it.
- Line 5: This registers the Help Topic as the default context help for message board objects. This is an optional attribute.
- Line 6: The relative path to the help file. Zope 3 will recognize file endings and create the appropriate filters for the output. Possible endings include `txt`, `rst`, and `html` and `stx`.
- Line 12–13: Here we register a topic specifically for the `review.html` view of a message in the messageboard.
- Line 11 & 19: Be careful to use URI-like syntax to specify the parent.

Now all you need to do is to restart Zope and go to a message board's message review view. Like all management pages, there a "Help" link on the very right side below the tabs. Usually this link just brings you to the generic online help screen, but if you click on it from the message board's review screen, you will see the help for this particular view. Another possibility would be to create special `Message` and `MessageBoard` object introduction screens, but I found this to be overkill in this situation.

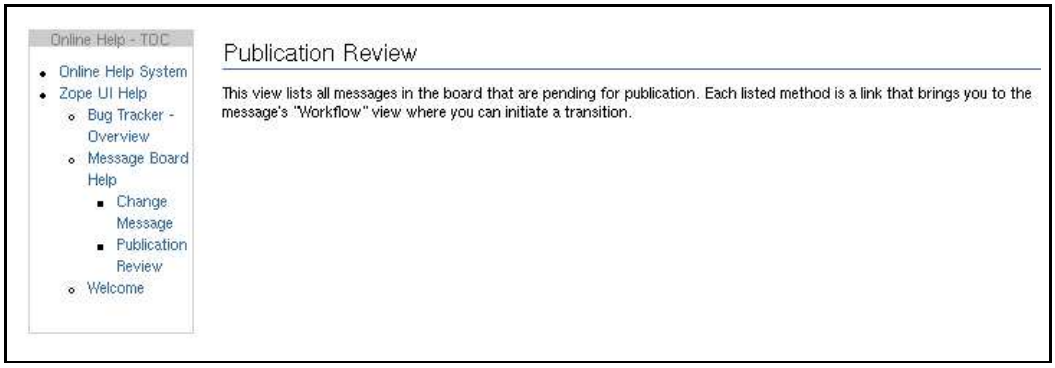


Figure 21.1: The help screen for the message board's "Review Messages" tab.

Exercises

1. Implement online help screens for all other views.

CHAPTER 22

OBJECT TO FILE SYSTEM MAPPING USING FTP AS EXAMPLE

Difficulty

Newcomer

Skills

- Be familiar with the Message Board Demo package up to this point.
- Good understanding of the Component Architecture, especially Adapters.
- Feel comfortable with writing ZCML-based configuration.
- Basic knowledge of the filesystem interfaces. Optional.

Problem/Task

Zope provides by default an FTP server , which is a filesystem based protocol. This immediately raises the question about the way objects are mapped to the filesystem representation and back. To accomplish the mapping in a flexible and exchangeable way, there exists a set of interfaces that can be implemented as adapters to provide a representation that the FTP Publisher understands. This chapter shows how to implement some of the interfaces for a custom filesystem representation.

Solution

At this point you might wonder: "Why in the world would we have to write our own filesystem support? Is Zope not providing any implementations by default?"

Well, to answer the latter question: Yes and no. There is an adapter registered for `IContainer` to `IReadDirectory` and `IWriteDirectory`. However, they are not very useful, since our Message Board and Message objects are not only containers but also contain content that is not displayed anywhere. Just start Zope 3 and check it out yourself. Thus it will be the goal of this chapter to create a representation that handles the containment and the content at the same time.

Since the core has already a lot of code presenting `IContainer` implementations as directories, we should reuse this part of the framework. The content of an object could be simply represented by a virtual file called `contents`, which contains all relevant data normalized as simple plain text. Note also that we will not need to have a complete mapping between the object and the filesystem representation, since we do not need or want to expose annotations for example. I suggest that the `contents` of the `MessageBoard` object simply contains the data of the `description` attribute and for the `Message` I propose the following format:

```
Title: <message title>

<message body>
```

This way we can also parse easily the title when new contents is submitted, since we want to implement the read *and* write interfaces of the filesystem representation. One of the main goals is to keep the `VirtualContentsFile` class as generic as possible, so that we can use it for both message boards and messages. To do that the virtual file must delegate the request to create the plain text representation to a component that knows about the specifics of the respective content object. For this task, we will have a simple `IPlainText` adapter that provides the plain text representation of each content component's contents.

22.1 Step I: Plain Text Adapters

As said above, `IPlainText` adapters are responsible to return and accept the plain text representation of the object's content and just do the right thing with the data. They are very simple objects, having two methods, one for providing and one for processing the text.

22.1.1 (a) The Interface

The interface is simple, it defines a `getText()` and a `setText()` method:

```
class IPlainText(Interface):
    """This interface allows you to represent an object's content in plain
    text."""

    def getText():
```

```
6         """Get a pure text representation of the object's content."""
7
8     def setText(text):
9         """Write the text to the object."""
```

This interface should be placed in the `interfaces` module of the messageboard. In the doc strings I refer to the object's "content" without further qualifying it. With content I mean all data (property values) that should be represented via the plain text representation.

The `setText()` method could in theory become quite complex, depending on how many properties you want to map and how you represent them. You will see that in our two cases it will be still fairly simple.

22.1.2 (b) The Implementation

We need two implementations, one for the `Message` and one for the `MessageBoard` class. Note that I skipped writing tests at this point, since the functionality of these adapters are carefully tested in the following code.

First, we write the message board adapter, so open the `messageboard.py` file and add the following code:

```
1 from book.messageboard.interfaces import IPlainText
2
3 class PlainText:
4
5     implements(IPlainText)
6
7     def __init__(self, context):
8         self.context = context
9
10    def getText(self):
11        return self.context.description
12
13    def setText(self, text):
14        self.context.description = unicode(text)
```

This is an extremely simple implementation of the `IPlainText` interface, since we map only one attribute.

▷ Line 14: Make sure that the incoming text is unicode, which is very important for the system's integrity.

The implementation for the Message (put it in `message.py`) looks like this:

```
1 from book.messageboard.interfaces import IPlainText
2
3 class PlainText:
4
5     implements(IPlainText)
6
7     def __init__(self, context):
```

```
8         self.context = context
9
10    def getText(self):
11        return 'Title: %s\n\n%s' %(self.context.title,
12                                   self.context.body)
13
14    def setText(self, text):
15        if text.startswith('Title: '):
16            title, text = text.split('\n', 1)
17            self.context.title = title[7:]
18
19        self.context.body = text.strip()
```

This implementation is more interesting, since we map two properties to the plain text.

▷ Line 11-12: In typical MIME-header style, define a field with the pattern ⟨`name`⟩`:` ⟨`value`⟩ for the title and place the body as content. Note that the standard requires an empty line after the headers too.

▷ Line 15-17: Check whether a title was specified in this upload. If so, extract the title from the data and store the title; if not just ignore the title altogether. Finally store the rest of the text as the body.

22.1.3 (c) The Configuration

The last step is to register the two adapters with the Component Architecture. Just add the following two `adapter` directives to `configure.zcml`:

```
<adapter
    factory=".messageboard.PlainText"
    provides=".interfaces.IPlainText"
    for=".interfaces.IMessageBoard" />

<adapter
    factory=".message.PlainText"
    provides=".interfaces.IPlainText"
    for=".interfaces.IMessage" />
```

We are now done. Even though we have two new fully-functional components at this point, we gained no new functionality yet, since these adapters are not used anywhere. We have to complete the next two sections to see any results.

22.2 Step II: The "Virtual Contents File" Adapter

How we implement the virtual `contents` file is fully up to us. However, there are benefits of choosing one way over another, since it will save us some work.

The best method is to create a new interface `IVirtualContentsFile`, which extends `zope.app.file.interfaces.IFile`. The advantage is that there are already filesystem-specific adapters (implementing `zope.app.filerepresentation.interfaces.IReadFile` and `zope.app.filerepresentation.interfaces.IWriteFile`) for the above mentioned interface. `IFile` might not be the best and most concise interface for our needs, but the advantages of using it are very convincing.

22.2.1 (a) The Interface

When you look through the Zope 3 source code, you will notice that the `IFile` and `IFileContent` interfaces go hand in hand with each. Thus, our virtual contents file interface will extend both of these interfaces.

```
from zope.app.file.interfaces import IFile, IFileContent

class IVirtualContentsFile(IFile, IFileContent):
    """Marker Interface to mark special Message and Message Board
    Contents files in FS representations."""
```

22.2.2 (b) The Implementation

Now the fun begins. First we note that `IFile` requires three properties, `contentType`, `data`, and `size`. While `data` and `size` are obvious, we need to think a bit about `contentType`. Since we really just want to return always `text/plain`, the accessor should statically return `text/plain` and the mutator should just ignore the input.

To make a long story short, here is the code, which you should place in a new file called `filerepresentation.py`:

```
from zope.interface import implements
from interfaces import IVirtualContentsFile, IPlainText

class VirtualContentsFile(object):

    implements(IVirtualContentsFile)

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

    def setContentType(self, contentType):
        '''See interface IFile'''
        pass

    def getContentType(self):
        '''See interface IFile'''
        return u'text/plain'

    contentType = property(getContentType, setContentType)

```

```
21      def edit(self, data, contentType=None):
22          '''See interface IFile'''
23          self.setData(data)
24
25      def getData(self):
26          '''See interface IFile'''
27          adapter = IPlainText(self.context)
28          return adapter.getText() or u''
29
30      def setData(self, data):
31          '''See interface IFile'''
32          adapter = IPlainText(self.context)
33          return adapter.setText(data)
34
35      data = property(getData, setData)
36
37      def getSize(self):
38          '''See interface IFile'''
39          return len(self.getData())
40
41      size = property(getSize)
```

▷ Line 11–13: As promised, the mutator ignores the input totally and is really just an empty method.

▷ Line 15–17: Make sure we *always* return "text/plain".

▷ Line 25–28 & 30–33: Now we are making use of our previously created `PlainText` adapters. We simply use the two available API methods.

This was pretty straightforward. There are really no surprises here.

22.2.3 (c) The Tests

Since even the last coding step did not provide a functional piece of code, it becomes so much more important to write some careful tests for the `VirtualContentsFile` component. Another requirement of the tests are that this adapter is being tested with both `MessageBoard` and `Message` instances. To realize this, we write an a base test and then realize this test for each component. So in the `tests` folder, create a new file called `test_filerepresentation.py` and add the following content:

```
1  import unittest
2  from zope.interface.verify import verifyObject
3  from zope.app import zapi
4  from zope.app.tests import ztapi
5  from zope.app.tests.placelesssetup import PlacelessSetup
6
7  from book.messageboard.interfaces import \
8       IVirtualContentsFile, IPlainText, IMessage, IMessageBoard
9    from book.messageboard.message import \
10      Message, PlainText as MessagePlainText
11 from book.messageboard.messageboard import \
```

```
12      MessageBoard, PlainText as MessageBoardPlainText
13 from book.messageboard.filerepresentation import VirtualContentsFile
14
15 class VirtualContentsFileTestBase(PlacelessSetup):
16
17     def _makeFile(self):
18         raise NotImplemented
19
20     def _registerPlainTextAdapter(self):
21         raise NotImplemented
22
23     def setUp(self):
24         PlacelessSetup.setUp(self)
25         self._registerPlainTextAdapter()
26
27     def testContentType(self):
28         file = self._makeFile()
29         self.assertEqual(file.getContentType(), 'text/plain')
30         file.setContentType('text/html')
31         self.assertEqual(file.getContentType(), 'text/plain')
32         self.assertEqual(file.contentType, 'text/plain')
33
34     def testData(self):
35         file = self._makeFile()
36
37         file.setData('Foobar')
38         self.assert_(file.getData().find('Foobar') >= 0)
39         self.assert_(file.data.find('Foobar') >= 0)
40
41         file.edit('Blah', 'text/html')
42         self.assertEqual(file.contentType, 'text/plain')
43         self.assert_(file.data.find('Blah') >= 0)
44
45     def testInterface(self):
46         file = self._makeFile()
47         self.failUnless(IVirtualContentsFile.providedBy(file))
48         self.failUnless(verifyObject(IVirtualContentsFile, file))
49
50
51 class MessageVirtualContentsFileTest(VirtualContentsFileTestBase,
52                                      unittest.TestCase):
53
54     def _makeFile(self):
55         return VirtualContentsFile(Message())
56
57     def _registerPlainTextAdapter(self):
58         ztapi.provideAdapter(IMessage, IPlainText, MessagePlainText)
59
60     def testMessageSpecifics(self):
61         file = self._makeFile()
62         self.assertEqual(file.context.title, '')
63         self.assertEqual(file.context.body, '')
64         file.data = 'Title: Hello\n\nWorld'
65         self.assertEqual(file.context.title, 'Hello')
66         self.assertEqual(file.context.body, 'World')
67         file.data = 'World 2'
68         self.assertEqual(file.context.body, 'World 2')
69
```

```
70
71 class MessageBoardVirtualContentsFileTest(
72       VirtualContentsFileTestBase, unittest.TestCase):
73
74     def _makeFile(self):
75         return VirtualContentsFile(MessageBoard())
76
77     def _registerPlainTextAdapter(self):
78         ztapi.provideAdapter(IMessageBoard, IPlainText,
79                              MessageBoardPlainText)
80
81     def testMessageBoardSpecifics(self):
82         file = self._makeFile()
83         self.assertEqual(file.context.description, '')
84         file.data = 'Title: Hello\n\nWorld'
85         self.assertEqual(file.context.description,
86                          'Title: Hello\n\nWorld')
87         file.data = 'World 2'
88         self.assertEqual(file.context.description, 'World 2')
89
90 def test_suite():
91     return unittest.TestSuite((
92         unittest.makeSuite(MessageVirtualContentsFileTest),
93         unittest.makeSuite(MessageBoardVirtualContentsFileTest),
94         ))
95
96 if __name__ == '__main__':
97     unittest.main(defaultTest='test_suite')
```

- ▷ Line 5: Since we are going to make use of adapters, we will need to bring up the component architecture using the `PlacelessSetup`.
- ▷ Line 7–13: Imports all the relevant interfaces and components for this test. This is always so much, since we have to register the components by hand (instead of ZCML).
- ▷ Line 17–18: The implementation of this method should create a `VirtualContentsFile` adapter having the correct object as `context`. Since the context varies, the specific test case class has to take of its implementation.
- ▷ Line 20–21: Since there is a specific adapter registration required for each case (board and message), we will have to leave that up to the test case implementation as well.
- ▷ Line 27–32: We need to make sure that the `plain/text` setting can never be overwritten.
- ▷ Line 34–43: We can really just make some marginal tests here, since the storage details really depend on the `IPlainText` implementation. There will be stronger tests in the specific test cases for the message board and message (see below).

▷ Line 45–48: Always make sure that the interface is completely implemented by the component.

▷ Line 51: This is the beginning of a concrete test case that implements the base test. Note that you should only make the concrete implementation a `TestCase`.

▷ Line 54–55: Just stick a plain, empty `Message` instance in the adapter.

▷ Line 60–68: Here we test that the written contents of the virtual file is correctly passed and the right properties are set.

▷ Line 71–88: Pretty much the same that was done for the `Message` test.

▷ Line 90–97: The usual test boilerplate.

You can now run the test and verify the functionality of the new tests.

22.2.4 (d) The Configuration

This section would not be complete without a registration. While we do not need to register the file representation component, we are required to make some security assertions about the object's methods and properties. I simply copied the following security assertions from the `File` content component's configuration.

```
1  <content class=".filerepresentation.VirtualContentsFile">
2
3    <implements interface="
4        zope.app.annotation.interfaces.IAttributeAnnotatable" />
5
6    <require
7        permission="book.messageboard.View"
8        interface="zope.app.filerepresentation.interfaces.IReadFile" />
9
10   <require
11       permission="zope.messageboard.Edit"
12       interface="zope.app.filerepresentation.interfaces.IWriteFile"
13       set_schema="zope.app.filerepresentation.interfaces.IReadFile" />
14
15 </content>
```

▷ Line 3–4: We need the virtual file to be annotable, so it can reach the DublinCore for dates/times and owner information.

22.3 Step III: The `IReadDirectory` implementation

After all the preparations are complete, we are finally ready to give our content components, `MessageBoard` and `Message`, a cool filesystem representation.

22.3.1 (a) The Implementation

The first fact we should notice is that `zope.app.filerepresentation.ReadDirectory` has already a nice implementation, except for the superfluous `SiteManager` and the missing `contents` file. So we simply take this class (subclass it) and merely overwrite `keys()`, `get(key,default=None)`, and `__len__()`. All other methods depend on these three. So our code for the ReadDirectory looks something like that (place in `filerepresentation.py`):

```
1  from zope.app.filerepresentation.interfaces import IReadDirectory
2  from zope.app.folder.filerepresentation import \
3       ReadDirectory as ReadDirectoryBase
4
5  class ReadDirectory(ReadDirectoryBase):
6      """An special implementation of the directory."""
7
8      implements(IReadDirectory)
9
10     def keys(self):
11         keys = self.context.keys()
12         return list(keys) + ['contents']
13
14     def get(self, key, default=None):
15         if key == 'contents':
16             return VirtualContentsFile(self.context)
17         return self.context.get(key, default)
18
19     def __len__(self):
20         l = len(self.context)
21         return l+1
```

- ▷ Line 10–12: When being asked for a list of names available for this container, we get the list of keys plus our virtual `contents` file.
- ▷ Line 14–17: All objects are simply found in the context (MessageBoard or Message) itself, except the `contents`. When the system asks for the `contents`, we simply give it a `VirtualContentsFile` instance that we prepared in the previous section and we do not have to worry about anything, since we know that the system knows how to handle `zope.app.file.interfaces.IFile` objects.
- ▷ Line 19–21: Obviously, we pretend to have one more object than we actually have.

Now we are done with our implementation. Let's write some unit tests to ensure the functionality and then register the filesystem components.

22.3.2 (b) The Tests

For testing the `ReadDirectory` implementation, we again need to test it with the `MessageBoard` and `Message` components. So similar to the previous tests, we have

a base test with specific implementations. Also note that it will not be necessary to test all `IReadDirectory` methods, since they are already tested in the base class tests. So we are just going to test the methods we have overridden:

```
from book.messageboard.filerepresentation import ReadDirectory

class ReadDirectoryTestBase(PlacelessSetup):

    def _makeDirectoryObject(self):
        raise NotImplemented

    def _makeTree(self):
        base = self._makeDirectoryObject()
        msg1 = Message()
        msg1.title = 'Message 1'
        msg1.description = 'This is Message 1.'
        msg11 = Message()
        msg11.title = 'Message 1-1'
        msg11.description = 'This is Message 1-1.'
        msg2 = Message()
        msg2.title = 'Message 1'
        msg2.description = 'This is Message 1.'
        msg1['msg11'] = msg11
        base['msg1'] = msg1
        base['msg2'] = msg2
        return ReadDirectory(base)

    def testKeys(self):
        tree = self._makeTree()
        keys = list(tree.keys())
        keys.sort()
        self.assertEqual(keys, ['contents', 'msg1', 'msg2'])
        keys = list(ReadDirectory(tree['msg1']).keys())
        keys.sort()
        self.assertEqual(keys, ['contents', 'msg11'])

    def testGet(self):
        tree = self._makeTree()
        self.assertEqual(tree.get('msg1'), tree.context['msg1'])
        self.assertEqual(tree.get('msg3'), None)
        default = object()
        self.assertEqual(tree.get('msg3', default), default)
        self.assertEqual(tree.get('contents').__class__,
                         VirtualContentsFile)

    def testLen(self):
        tree = self._makeTree()
        self.assertEqual(len(tree), 3)
        self.assertEqual(len(ReadDirectory(tree['msg1'])), 2)
        self.assertEqual(len(ReadDirectory(tree['msg2'])), 1)


class MessageReadDirectoryTest(ReadDirectoryTestBase,
                               unittest.TestCase):

    def _makeDirectoryObject(self):
        return Message()

```

```
55
56 class MessageBoardReadDirectoryTest(ReadDirectoryTestBase,
57                                      unittest.TestCase):
58
59     def _makeDirectoryObject(self):
60         return MessageBoard()
```

▷ Line 5–6: Return an instance of the object to be tested.

▷ Line 8–22: Create an interesting message tree on top of the base. This will allow some more detailed testing.

▷ Line 24–31: Make sure this `contents` file and the sub-messages are correctly listed.

▷ Line 33–40: Now let's also make sure that the objects we get are the right ones.

▷ Line 42–46: A simple test for the number of contained items (including the `contents`).

▷ Line 49–60: The concrete implementations of the base test. Nothing special.

After you are done writing the tests, do not forget to add the two new `TestCases` to the `TestSuite`.

22.3.3 (c) The Configuration

Finally we simply register our new components properly using the following ZCML directives:

```
1  <adapter
2     for=".interfaces.IMessageBoard"
3     provides="zope.app.filerepresentation.interfaces.IReadDirectory"
4     factory=".filerepresentation.ReadDirectory"
5     permission="zope.View"/>
6
7  <adapter
8     for=".interfaces.IMessage"
9     provides="zope.app.filerepresentation.interfaces.IReadDirectory"
10    factory=".filerepresentation.ReadDirectory"
11    permission="zope.View"/>
```

That's it. You can now restart Zope and test the filesystem representation with an FTP client of your choice. In the following sequence diagram you can see how a request is guided to find its information and return it properly.

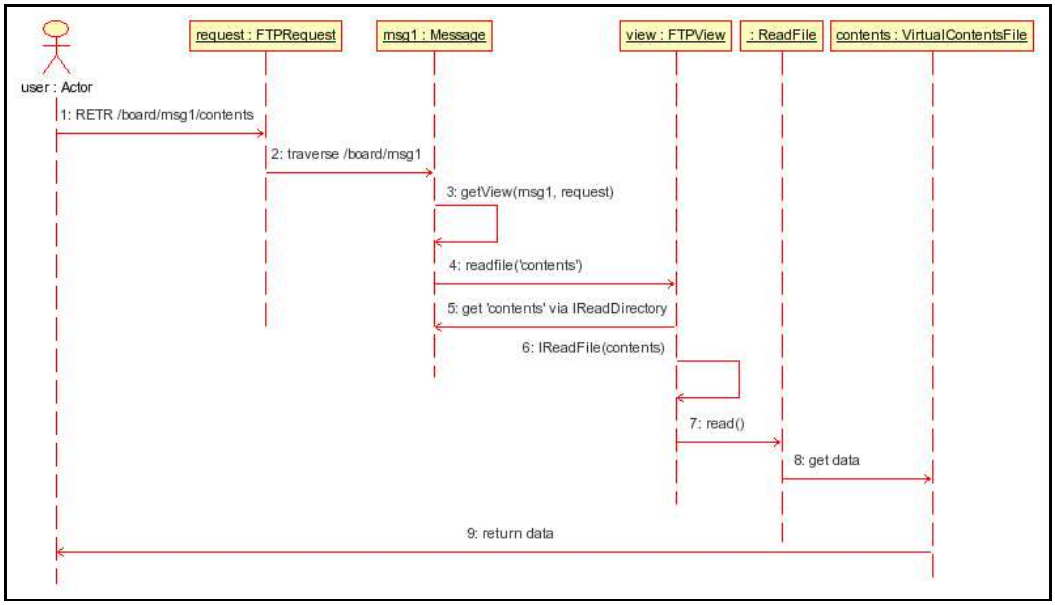


Figure 22.1: Collaboration diagram of the inner working from requesting the `contents` "file" to receiving the actual data.

22.4 Step IV: The Icing on the Cake – A special Directory Factory

While you were playing around with the new filesystem support, you might have tried to create a directory to see what happened and it probably just caused a system error, since no adapter was found from `IMessage`/ `IMessageBoard` to `IDirectoryFactory`. Since such a behavior is undesirable, we should create a custom adapter that provides `IDirectoryFactory`. The `IWriteDirectory` adapter of any container object then knows how to make use of this factory adapter. So we add the following trivial factory to our filesystem code:

```
1  from zope.app.filerepresentation.interfaces import IDirectoryFactory
2  from message import Message
3
4  class MessageFactory(object):
5      """A simple message factory for file system representations."""
6
7      implements(IDirectoryFactory)
8
9      def __init__(self, context):
10         self.context = context
11
12     def __call__(self, name):
13         """See IDirectoryFactory interface."""
```

```
14         return Message()
```

Registering the factory is just a matter of two adapter directives (one for each content component):

```
1  <adapter
2     for=".interfaces.IMessageBoard"
3     provides="zope.app.filerepresentation.interfaces.IDirectoryFactory"
4     factory=".filerepresentation.MessageFactory"
5     permission="zope.View" />
6
7  <adapter
8     for=".interfaces.IMessage"
9     provides="zope.app.filerepresentation.interfaces.IDirectoryFactory"
10    factory=".filerepresentation.MessageFactory"
11    permission="zope.View" />
```

Now we have finally made it. The filesystem support should be very smooth and usable at this point. You should be able to view all relevant content, upload new `contents` data and create new messages. The only problem that might remain is that some FTP clients (like KDE's FTP support) try to upload the `contents` file as `contents.part` and then rename it to `contents`. Since our filesystem code does not support such a feature, this will cause an error; see exercise 2 for details.

Exercises

1. Currently there is no creation/modification date/time or creator defined for the virtual contents file. This is due to the fact that the respective Dublin Core properties were not set. The virtual file should really receive the same Dublin Core the `MessageBoard` or `Message` has. The easiest way would be simply to copy the Dublin Core annotation. Do that and make sure the data and the user are shown correctly.

2. In the final remarks of this chapter I stated that the current code will not work very well with some clients, such as Konqueror. Fix the code to support this behavior. The best would be to store the temporary file in an annotation and delete it, once it was moved.

CHAPTER 23

AVAILABILITY VIA XML-RPC

Difficulty

Sprinter

Skills

- Be familiar with the Message Board demo package up to this point.
- Feel comfortable with writing ZCML-based configuration.
- Some insight in the publisher framework. Optional.

Problem/Task

A very common way to communicate with remote systems is via XML-RPC, a very light-weight protocol on top of HTTP. Zope's HTTP server comes by default with a XML-RPC component. If we want to allow other systems to communicate with our message board, then we need to declare the methods that will be available via XML-RPC.

Solution

You might wonder at this point why we don't simply map all the existing methods to XML-RPC and be done with it. There are three reasons for not doing this. First, XML-RPC handles only a limited amount of data types. In the following table you see a mapping of Python types to XML-RPC data type elements:

- `integer` $\longleftrightarrow$ $\langle$`int`$\rangle$

- `float` ⟷ ⟨`double`⟩
- `string` ⟷ ⟨`string`⟩
- `list` ⟷ ⟨`array`⟩
- `dict/struct` ⟷ ⟨`dict`⟩
- `bool` ⟷ ⟨`boolean`⟩
- `xmlrpclib.Binary` ⟷ ⟨`binary`⟩
- `xmlrpclib.DateTime` ⟷ ⟨`dateTime`⟩

As you can see, there is no support for `None` and unicode, which are huge drawbacks for XML-RPC in general.

Second, another disadvantage is the lack of keyword arguments. XML-RPC only understands regular positional arguments and arguments with default values. Third, since Python returns `None` by default for methods, all methods that do not have a return value are doomed.

Now that we have briefly discussed the shortcomings of XML-RPC, we should look ahead and say that XML-RPC is still a very powerful tool that can be used without much hassle.

23.1 Step I: Creating "Methods" XML-RPC Presentation Components

Obviously we have to create a view component for both content objects, `MessageBoard` and `Message`. However, both share the functionality that they can contain and manage sub-messages, so that it is desired to factor out this functionality into a common base class, `MessageContainerMethods`. Since we want to keep the XML-RPC code separate, create a new module called `xmlrpc.py` in the `messageboard` directory and add the following content:

```
1  from zope.event import notify
2  from zope.app.publisher.xmlrpc import MethodPublisher
3
4  from zope.app.event.objectevent import ObjectCreatedEvent
5
6  from book.messageboard.message import Message
7
8  class MessageContainerMethods(MethodPublisher):
9
10   def getMessageNames(self):
11       """Get a list of all messages."""
12       return list(self.context.keys())
13
```

```
14   def addMessage(self, name, title, body):
15       """Add a message."""
16       msg = Message()
17       msg.title = title
18       msg.body = body
19       notify(ObjectCreatedEvent(msg))
20       self.context[name] = msg
21       return name
22
23   def deleteMessage(self, name):
24       """Delete a message. Return True, if successful."""
25       self.context.__delitem__(name)
26       return True
```

▷ Line 8: Make the class a `MethodPublisher`, which is similar to a `BrowserView` and its constructor will expect a `context` object and a `XMLRPCRequest` instance.

▷ Line 10–12: Return a list of all message names. But remember, we implemented the containment with BTrees (see first chapter of this part), so that `keys()` will not return a list or tuple as a result, but a btree-items object. Therefore we need to implicitly cast the result to become a list, so that XML-RPC is able understand it.

▷ Line 19: Since we want to play nice with the system, let it know that we created a new content object.

▷ Line 21, 26: Make sure we return something, so that XML-RPC will not be upset with us. As mentioned before, if no return value is specifed, `None` is implicitly returned, which XML-RPC does nto understand.

The two actual views for our content objects are now also implementing accessor and mutator methods for their properties. So here are the two views:

```
1  from zope.app.event.objectevent import ObjectModifiedEvent
2
3  class MessageMethods(MessageContainerMethods):
4
5      def getTitle(self):
6          return self.context.title
7
8      def setTitle(self, title):
9          self.context.title = title
10         notify(ObjectModifiedEvent(self.context))
11         return True
12
13     def getBody(self):
14         return self.context.body
15
16     def setBody(self, body):
17         self.context.body = body
18         notify(ObjectModifiedEvent(self.context))
```

```
19         return True
20
21
22 class MessageBoardMethods(MessageContainerMethods):
23
24     def getDescription(self):
25         return self.context.description
26
27     def setDescription(self, description):
28         self.context.description = description
29         notify(ObjectModifiedEvent(self.context))
30         return True
```

▷ Line 10, 18 & 29: When modifying a message board or message, we have to explicitly send out a modification notification event. We did not have to deal with this until now, since for browser forms these events are created automatically by the forms machinery.

▷ Line 11, 19 & 30: Again, we need to make sure we do not just return `None` from a method.

That's already everything from a coding point of perspective. But before we hook up the code in the component architecture, we need to do some testing.

23.2 Step II: Testing

Of course, the testing code is multiples more complex than the actual implementation, since we have to bring up the component architecture and the event service manually. Similar to the implementation, we can again separate the container-related tests in a base class (the code should be located in `tests/test_xmlrpc.py`):

```
1 from zope.app import zapi
2 from zope.app.tests.placelesssetup import PlacelessSetup
3
4 class MessageContainerTest(PlacelessSetup):
5
6     def _makeMethodObject(self):
7         return NotImplemented
8
9     def _makeTree(self):
10         methods = self._makeMethodObject()
11         msg1 = Message()
12         msg1.title = 'Message 1'
13         msg1.description = 'This is Message 1.'
14         msg2 = Message()
15         msg2.title = 'Message 1'
16         msg2.description = 'This is Message 1.'
17         methods.context['msg1'] = msg1
18         methods.context['msg2'] = msg2
19         return methods
```

```
20
21      def test_getMessageNames(self):
22          methods = self._makeTree()
23          self.assert_(isinstance(methods.getMessageNames(), list))
24          self.assertEqual(list(methods.context.keys()),
25                           methods.getMessageNames())
26
27      def test_addMessage(self):
28          methods = self._makeTree()
29          self.assertEqual(methods.addMessage('msg3', 'M3', 'MB3'),
30                           'msg3')
31          self.assertEqual(methods.context['msg3'].title, 'M3')
32          self.assertEqual(methods.context['msg3'].body, 'MB3')
33
34      def test_deleteMessage(self):
35          methods = self._makeTree()
36          self.assertEqual(methods.deleteMessage('msg2'), True)
37          self.assertEqual(list(methods.context.keys()), ['msg1'])
```

- ▷ Line 6–7: The implementation of this method should return a valid XML-RPC method publisher.
- ▷ Line 9–19: Create an interesting message tree, so that we have something to test with.
- ▷ Line 21–25: Make sure the names list is converted to a Python list and all elements are contained in it.
- ▷ Line 27–32: This method obviously tests the adding capability. We just try to make sure that the correct attributes are assigned to the message.
- ▷ Line 34–37: Simply checks that a message is really deleted.

Now that we have the base class, we can implement the real test cases and add tests for the property accessors and mutators:

```
1  import unittest
2
3  from zope.publisher.xmlrpc import TestRequest
4
5  from book.messageboard.message import Message
6  from book.messageboard.messageboard import MessageBoard
7  from book.messageboard.xmlrpc import MessageBoardMethods, MessageMethods
8
9  class MessageBoardMethodsTest(MessageContainerTest, unittest.TestCase):
10
11      def _makeMethodObject(self):
12          return MessageBoardMethods(MessageBoard(), TestRequest())
13
14      def test_description(self):
15          methods = self._makeTree()
16          self.assertEqual(methods.getDescription(), '')
17          self.assertEqual(methods.setDescription('Board 1') , True)
18          self.assertEqual(methods.getDescription(), 'Board 1')
```

```
19
20 class MessageMethodsTest(MessageContainerTest, unittest.TestCase):
21
22     def _makeMethodObject(self):
23         return MessageMethods(Message(), TestRequest())
24
25     def test_title(self):
26         methods = self._makeTree()
27         self.assertEqual(methods.getTitle(), '')
28         self.assertEqual(methods.setTitle('Message 1') , True)
29         self.assertEqual(methods.getTitle(), 'Message 1')
30
31     def test_body(self):
32         methods = self._makeTree()
33         self.assertEqual(methods.getBody(), '')
34         self.assertEqual(methods.setBody('Body 1') , True)
35         self.assertEqual(methods.getBody(), 'Body 1')
36
37
38 def test_suite():
39     return unittest.TestSuite((
40         unittest.makeSuite(MessageBoardMethodsTest),
41         unittest.makeSuite(MessageMethodsTest),
42         ))
43
44 if __name__ == '__main__':
45     unittest.main(defaultTest='test_suite')
```

▷ Line 11–12 & 22–23: Create a XML-RPC method publisher for the message board and the message, respectively. To do that we need an object instance (no problem) and an XML-RPC request. Luckily, like for the browser publisher, the XML-RPC publisher provides a `TestRequest` which was written for its easy usage in unit tests like these.

▷ Line 38–45: And again the usual unit test boiler plate.

The rest of the code is not so interesting and should be obvious to the reader. Please run these tests now and make sure that everything passes.

23.3 Step III: Configuring the new Views

To register the XML-RPC views, you need to import the `xmlrpc` namespace into your main configuration file using

```
1 xmlns:xmlrpc="http://namespaces.zope.org/xmlrpc"
```

in the `zopeConfigure` element. Now you simply add the following two directives to the configuration:

```
1 <xmlrpc:view
2     for=".interfaces.IMessageBoard"
```

```
3     permission="book.messageboard.Edit"
4     methods="getMessageNames addMessage deleteMessage
5              getDescription setDescription"
6     class=".xmlrpc.MessageBoardMethods" />
7
8  <xmlrpc:view
9     for=".interfaces.IMessage"
10    permission="book.messageboard.Edit"
11    methods="getMessageNames addMessage deleteMessage
12             getTitle setTitle getBody setBody"
13    class=".xmlrpc.MessageMethods" />
```

- ▷ Line 2: This view is for `IMessageBoard` objects.
- ▷ Line 3: XML-RPC views require the `book.messageboard.Edit` permission, which means that someone has to authenticate before using these methods.
- ▷ Line 4–5: This is the list of methods that will be available as XML-RPC methods on the messageboard.
- ▷ Line 6: The method publisher class is `.xmlrpc.MessageBoardMethods`, which provides the previously defined methods.
- ▷ Line 8–13: Repeat the previous precedure for `IMessage` components.

Now you can restart Zope 3 and give your XML-RPC methods a run. But oh no, how do we test this? Certainly a browser will not get us very far.

23.4 Step IV: Testing the Features in Action

Python has a really nice XML-RPC module that can be used as XML-RPC client and also provides some extra help with basic authentication. In order to save the reader some typing, I have provided a module called `xmlrpc_client.py` in the `messageboard` package which you can call from the command line:

```
1 # ./xmlrpc_client.py
```

You will be asked for the URL to your messageboard object, your username and password. Once this is done, you are presented with a normal Python prompt, except that there is a local variable called `board` available, which represents your XML-RPC connection. You can now use the available methods to manipulate the board at your heart's content.

```
# ./messageboard/xmlrpc_client.py
Message Board URL [http://localhost:8080/board/]:
Username: srichter
Password: ........
The message board is available as 'board'
Python 2.3 (#2, Aug 31 2003, 17:27:29)
[GCC 3.3.1 (Mandrake Linux 9.2 3.3.1-1mdk)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> board.getDescription()
'First Message Board'
>>> board.getMessageNames()
['msg1', 'msg2']
>>> board.msg1.getMessageNames()
['msg11']
>>> board.msg1.msg11.getTitle()
'Message 1-1'
>>> board.msg1.msg11.getBody()
'This is the first response to Message 1.'
>>> board.msg1.addMessage('msg12', 'Message 1-2', 'Second response!')
'msg12'
>>> board.msg1.getMessageNames()
['msg11', 'msg12']
>>> board.msg1.msg12.getTitle()
'Message 1-2'
>>> board.msg1.deleteMessage('msg12')
True
>>> board.msg1.getMessageNames()
['msg11']
>>>
```

Figure 23.1: A sample XML-RPC session using the provided client.

Exercises

1. We only made a very limited amount of the message board's functionality available via XML-RPC. Write a new XML-RPC method publisher to

   (a) manage mail subscriptions for messages.

   (b) manage the workflows of messages.

2. Due to the simple implementation of this chapter, message attachments are treated like messages when displaying message names. Improve the implementation and the API to handle attachments properly. This includes writing a method publisher for `IFile` components.

CHAPTER 24

DEVELOPING NEW SKINS

Difficulty

Newcomer

Skills

- Be familiar with the Message Board Demo package up to this point.
- Feel comfortable with writing ZCML-based view configuration. Optional.

Problem/Task

Until now we have only enhanced the messageboard by features, but have not done much to improve the user interface. In fact, we are still using the ZMI to do all our message board management, which is totally inappropriate to the end user. Therefore, this chapter will concentrate on developing a skin specifically designed for the message board that implements a user interface as it is typically seen in real world message board applications. While this package has little to do with Python development, I feel that it is a good final task for our two Content Component parts.

Solution

Skins (the equivalence of CMF Skins in Zope 2) are a method to implement a custom look and feel to existing views. This is very similar to HTML and CSS (Cascading Style Sheets), where the HTML code are equivalent to views (page templates, view classes) and the style sheets (CSS) are the skin over the HTML structural elements. Skins however, have another abstraction layer beneath.

A skin is really just a stack of layers. Each layer can contain any amount of views and resources. This allows particular views and resources to be overridden. For example, our style sheet (CSS) might have been defined in the `default` layer. However, this style sheet is really simplistic and inappropriate for our needs. We can then create a new layer `board` and place a new style sheet in it. Once that is done, we define a skin that places the `board` layer after the default layer, and all the new style definitions will be adopted.

24.1 Step I: Preparation

Before we can create our new skin, we need to make some preparations. In order not to confuse the original views and resources with the new ones, we create a package called `skin` in the `messageboard/browser` directory; do not forget to make a `__init__.py` file. Then create an empty `configure.zcml` file:

```
<configure
    xmlns="http://namespaces.zope.org/browser">

</configure>
```

Now hook up this configuration file from the browser's `configure.zcml` using:

```
<include package=".skin" />
```

24.2 Step II: Creating a New Skin

Creating a new skin is very easy and can be accomplished purely with ZCML configuration directives. The `browser` namespace has a special directive called `skin` that let's us do this, so add the following directive to the configuration file of the skin package:

```
<layer name="board"/>

<skin name="board" layers="board rotterdam default" />
```

The first directive creates a new layer, in which we will place all the new templates and which will make the skin unique. The second directive creates a skin named `board` that consists of a three element layer stack. The lowest layer is `default` which is overridden by `rotterdam`, which is overridden by `board`. Every browser presentation directive supports a `layer` attribute, which defines the layer in which a view or resource is placed. If no layer was specified, the presentation component is placed in the "default" skin.

You might also wonder why the `rotterdam` layer is placed here. The `rotterdam` layer contains some nice definitions, like the favicon and some other view code that is useful for us here as well. Other than that, we will not be using this layer actively.

24.3 Step III: Customizing the Base Templates

The first task is always to override the `skin_macros` and the `dialog_macros`. Usually the skin macros are defined by a file called `template.pt` and the dialog macros in `dialog_macros.pt`. Our new `template.pt` file might look something like this:

```
<metal:block define-macro="page">
  <metal:block define-slot="doctype">
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
  </metal:block>

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

  <head>
    <title metal:define-slot="title">Message Board for Zope 3</title>

    <style type="text/css" media="all"
       tal:content=
       "string: @import url(${context/++resource++board.css});">
      @import url(board.css);
    </style>

    <meta http-equiv="Content-Type"
        content="text/html;charset=utf-8" />

    <link rel="icon" type="image/png"
          tal:attributes="href context/++resource++favicon.png" />
  </head>

  <body>

    <div id="board_header" i18n:domain="messageboard">
      <img id="board_logo"
          tal:attributes="src context/++resource++logo.png" />
      <div id="board_greeting"> 
         <span i18n:translate="">Zope 3 Message Board</span>
      </div>
    </div>

    <div id="workspace">

      <div metal:define-slot="message" id="message"></div>

      <div id="content">
        <metal:block define-slot="body">
       This is the content.
        </metal:block>
      </div>

    </div>

    <div id="footer">

      <div id="actions">
```

```
50          <metal:block define-slot="actions" />
51        </div>
52        <div id="credits" i18n:domain="messageboard">
53          Powered by Zope 3.<br>
54          Stephan Richter in 2003
55        </div>
56      </div>
57
58    </body>
59
60  </html>
61
62  </metal:block>
```

- ▷ Line 12–16: Instead of the standard `zope3.css` we going to use a new `board.css` style sheet.
- ▷ Line 21–22: This favicon is provided by the `rotterdam` skin.
- ▷ Line 27–33: Do you see how simple the header is? A couple of styles, a logo, and a simple title will do for us.
- ▷ Line 47–56: The footer consists simply of a placeholder (slot) where we can later drop actions into and a tiny credits section.

There is not really much to this template. Notice how much simpler this is than for example the Rotterdam equivalent that can be found at `src/zope/app/rotterdam`. Similarly simple is the `dialog_macros.pt` page template, which you can find in the example code.

In the above template we referred to two new resources, the `logo.png` and the `board.css`. Both are configured as follows:

```
1  <resource
2      name="board.css" file="board.css" layer="board" />
3
4  <resource
5      name="logo.png" file="logo.png" layer="board" />
```

Note how the resource directive has a `layer` attribute to specify the layer. The initial CSS file (`board.css`) looks like this:

```
1  body {
2      font-family: Verdana, Arial, Helvetica, sans-serif;
3      background: white;
4      color: black;
5      margin: 0;
6      padding: 0pt;
7  }
8
9  h1, h2, h3, h4, h5, h6 {
10     font-weight: bold;
11     color: black;
```

```
}

/* Different headers are used for the same purpose,
   so make them all equal. */

h1, h2, h3 {
    font-size: 20pt;
    margin-top: 0px;
    margin-bottom: .8em;
    border-bottom: solid 1px #1E5ADB;
}

...

/* Header Stuff */

#board_header {
    background: #EEEEEE;
    border: solid 1px #AAAAAA;
    padding: 3pt;
    clear: both;
}

...

/* Footer stuff */

#footer {
    background: #EEEEEE;
    border: solid 1px #AAAAAA;
    padding: 0.5em;
    font-size: 85%;
}

...
```

For the full style sheet, see the example code. The templates are then registered for the `board` layer as follows:

```
<page
    for="*"
    name="skin_macros"
    permission="zope.View"
    layer="board"
    template="template.pt" />

<page
    for="*"
    name="dialog_macros"
    permission="zope.View"
    layer="board"
    template="dialog_macros.pt" />
```

▷ Line 2 & 9: The star means that this page is available for all objects.

▷ Line 5 & 12: The additional `layer` attribute is enough to specify the layer.

24.4 Step IV: Adding a Message Board Intro Screen

The simplest view of the message board is some sort of introduction screen for the message board, since it is just a simple page template. The template looks like this:

```
<html metal:use-macro="views/standard_macros/page">
  <body>
    <div id="content" metal:fill-slot="body"
        i18n:domain="messageboard">

      <h2 i18n:translate="">Welcome to the Message Board</h2>

      <p class="board_description" tal:content="context/description">
        Description of the Message Board goes here.
      </p>

      <div id="login_link">
        <a href="./posts.html">Click here to enter the board.</a>
      </div>

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

Place the template in the `skin` directory having the name `board_intro.pt`. The view must be registered for the layer using:

```
<page
    for="book.messageboard.interfaces.IMessageBoard"
    name="intro.html"
    permission="book.messageboard.View"
    layer="board"
    template="board_intro.pt" />
```

When you restart Zope 3 now, you should be able to reach this view using `http://localhost:8080/++skin++board/board/@@intro.html` assuming that the `MessageBoard` instance is called `board` and lives in the root folder.

24.5 Step V: Viewing a List of all Message Board Posts

Once the user enters the message board, s/he should be represented with a list of all top-level messages. In the actions section, the user should have an option to add a new message to the board and administrators can go to a review screen to publish messages that are still pending. You can find the source of the template in `board_posts.pt` in the `skin` directory of the message board example code.

The view is configured with the following instruction:

```
<page
    for="book.messageboard.interfaces.IMessageBoard"
    name="posts.html"
    permission="book.messageboard.View"
    layer="board"
```

Figure 24.1: Message Board introduction screen of the "board" skin.

```
6     class=".views.Posts"
7     template="board_posts.pt" />
```

As you can see, the page uses a view class which is located in `views.py`. The template `board_posts.pt` uses a method of the view that returns a list containing a dictionary with detailed information for each message. However, this does nothing new. We have functionality like this in the existing views, so we will not explain it here.

You should now be able to see a list of posts of the messageboard (but only those who are in the "published" workflow state).

24.6 Step VI: Adding a Post to the Message Board

When looking at the posts screen, you might already have clicked already on the "Create New Post" link and added a new message. You will have noticed that the add form did not bring you back to the posts overview but some management screen. To change that, we have to create a customized add form screen that specifies the return URL. The implementation in the `views.py` module looks as like that:

```
1 class AddMessage:
2     """Add-Form supporting class."""
3
4     def nextURL(self):
5         return '../@@posts.html'
```

This was straightforward.

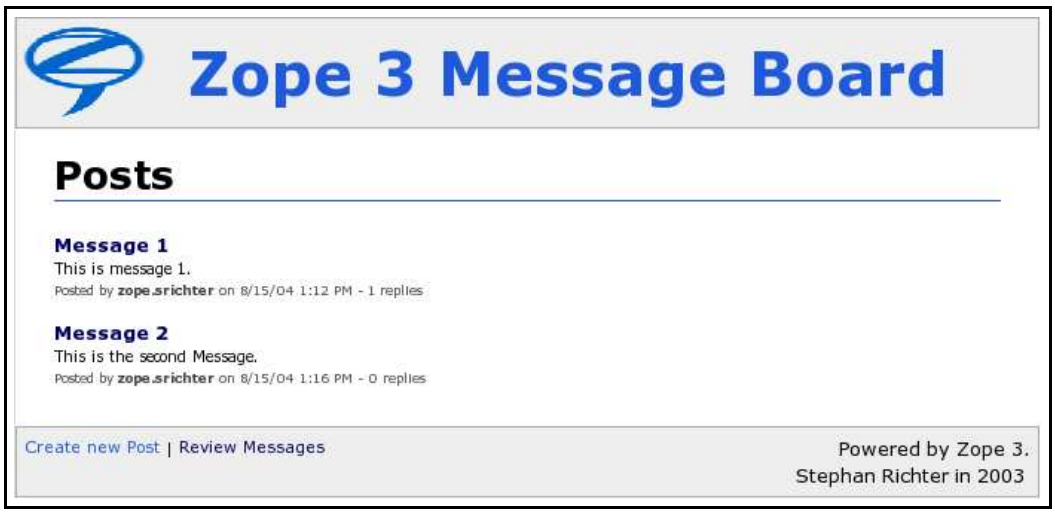


Figure 24.2: Posted messages of the message board.

By default you can enter a message id for the message. This is undesirable for our user-friendly skin. You may have noticed that you can leave the field empty, in which case the message id will be something like "Message" or "Message-2". That's because there is a mechanism that creates these generic names. The mechanism is used by the add form, if no name was specified. You can tell the add form to always use this mechanism for the `MessageBoard` and `Message` components by having them implement the `IContainerNamesContainer` interface. You can do this by adding the following directive to the `zope:content` directive of both objects in the main `configure.zcml` file:

```
<implements
    interface="zope.app.container.interfaces.IContainerNamesContainer"
    />
```

Finally we have to configure the new add form using

```
<addform
    label="Add Message"
    name="AddMessage.html"
    schema="book.messageboard.interfaces.IMessage"
    content_factory="book.messageboard.message.Message"
    permission="book.messageboard.Add"
    class=".views.AddMessage"
    layer="board"/>
```

This is not very hard either, right? After you restarted Zope, you can now admire the correct functionality of the add form.

24.7 Step VII: Reviewing "pending" Messages

Did you try to add a message? But where did it go? Well, remember that only published messages are visible.

While we have a message review screen for the management interface, it is not very usable. So we develop a much more simplified functionality in the `board` skin that can only publish messages. The following template displays all the pending messages and provides a checkbox for the moderator to select whether this message is about to be published (`board_review.pt`):

```
<html metal:use-macro="views/standard_macros/page">
  <body>
    <div metal:fill-slot="body" i18n:domain="messageboard">

      <h2 i18n:translate="">Review Pending Posts</h2>

      <form action="updateStatus.html" method="POST">

        <div id="message_line"
            tal:repeat="post view/getPendingMessagesInfo">
          <input type="checkbox" name="messages" value=""
            tal:attributes="value post/path" />
          <a href="" tal:attributes="href post/url"
            tal:content="post/title">Message Title</a>
          <div style="font-size: 70%">
            (Posted by <b tal:content="post/creator">Creator</b>
            on <b tal:replace="post/created">2003/01/01</b>)
          </div>
        </div>
        <br />
        <input type="submit" value="Publish"
            i18n:attributes="value" />

      </form>

    </div>
    <div id="actions" metal:fill-slot="actions">
      <a href="posts.html" i18n:translate="">View Posts</a>
    </div>
  </body>
</html>
```

In the corresponding Python view class, we can simply reuse the code we developed for the management version:

```
from zope.app.import zapi
from zope.app.dublincore.interfaces import ICMFDublinCore
from zope.app.interfaces.workflow import IProcessInstanceContainer

from book.messageboard.browser.messageboard import ReviewMessages

class Review(ReviewMessages):
    """Review messages for publication."""

    def getPendingMessagesInfo(self):
```

```
11          """Get all the display info for pending messages"""
12          msg_infos = []
13          for msg in self.getPendingMessages(self.context):
14              dc = ICMFDublinCore(msg)
15              info = {}
16              info['path'] = zapi.getPath(msg)
17              info['title'] = msg.title
18              info['creator'] = dc.creators[0]
19              formatter = self.request.locale.dates.getFormatter(
20                  'dateTime', 'medium')
21              info['created'] = formatter.format(dc.created)
22              info['url'] = zapi.getView(
23                  msg, 'absolute_url', self.request)() + \
24                  '/@@details.html'
25              msg_infos.append(info)
26          return msg_infos
27
28      def updateStatus(self, messages):
29          """Upgrade the stati from 'pending' to 'published'."""
30          if not isinstance(messages, (list, tuple)):
31              messages = [messages]
32
33          for path in messages:
34              msg = zapi.traverse(self.context, path)
35
36              adapter = IProcessInstanceContainer(msg)
37              adapter['publish-message'].fireTransition('pending_published')
38
39          return self.request.response.redirect('@@review.html')
```

▷ Line 5, 7, & 13: We are going to reuse some code from the other implementation.

▷ Line 28–39: This is the interesting method, since it actually fires the transition from "pending" to "published" status.

- Line 33–34: Since we were clever, we passed the path as checkbox value, so that we can now simply traverse to it.
- Line 36–37: Once we have the message, we get its process container and fire the transition.

The new review view can now be registered using the `pages` directive:

```
1 <pages
2     for="book.messageboard.interfaces.IMessageBoard"
3     class=".views.Review"
4     permission="book.messageboard.PublishContent"
5     layer="board">
6     <page name="review.html" template="board_review.pt"/>
7     <page name="updateStatus.html" attribute="updateStatus"/>
8 </pages>
```

Now restart Zope 3. Before we can enjoy publishing messages, we need to automate the transition from the "private" to "pending" status. To do that, go to the

"default" Site-Management folder, and from there to your "publish-message" workflow definition. There you will find an option called "Manage Transitions". Click on it and choose the transition named "private_pending". In the following edit form, there is a "Trigger Mode" option that is set to "Manual". Set it to "Automatic". This will cause the messages to move automatically from "initial" to "pending". If you now create a message, it will be automatically placed on the review list, from which you can then publish it as messageboard moderator. If you do not want any workflow at all, you can also set the "pending_published" "Trigger Mode" to "Automatic" and all newly created messages will automatically be published.

Figure 24.3: Review of all pending messages.

24.8 Step VIII: View Message Details

While the ZMI-based Message Details screen was very clear and detailed, it is not suitable for the end user. This view implements a better version of that screen and also adds the replies thread at the end. One of the available actions is to reply to the message. This is the screen we will look at next.

Since there is nothing new or very interesting going on in the code, I am going to skip over it and move directly to the next one. You will find the code in the accompanying package as usual.

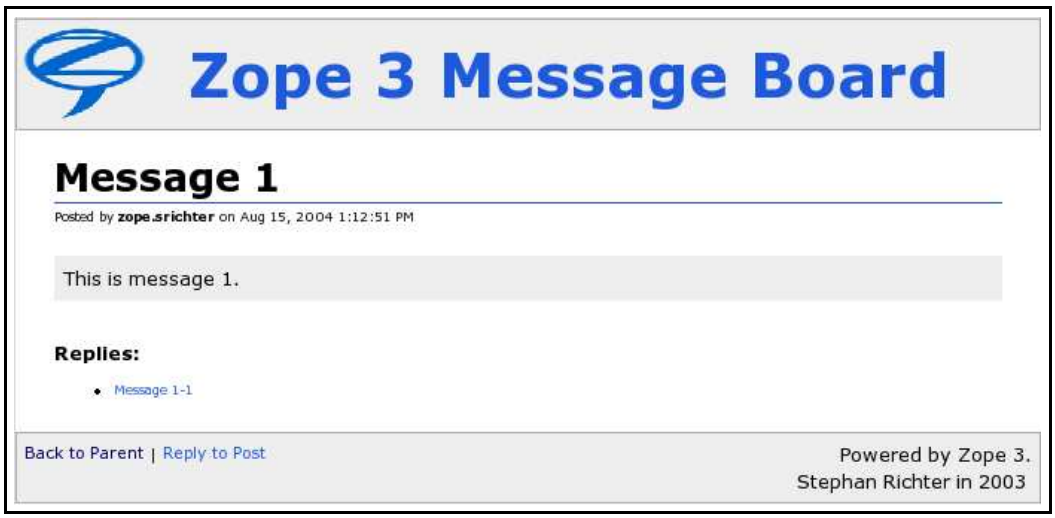


Figure 24.4: Display of all details of a message.

24.9 Step IX: Replies to Messages

Wow, another add form; can we do this? Yes, of course. The "Reply to Message" add form is a bit special though, since it should contain dynamic default values. For example, if the original message had a title called "My Title", then the suggested title for the new reply message should be "Re: My Title". Similarly, every text line of the original message body should be prefixed with "¿ ".

The best place I found to insert the dynamic default values is in the private `_setUpWidgets()` methods, which creates a data dictionary that is passed to the widget instantiater. The code for the entire add form view class looks like that:

```
1  from zope.app.form.interfaces import IInputWidget
2  from zope.app.form.utility import setUpWidgets
3
4  class ReplyMessage:
5      """Add-Form supporting class."""
6
7      def nextURL(self):
8          return '../@@details.html'
9
10     def _setUpWidgets(self):
11         """Alllow addforms to also have default values."""
12         parent = self.context.context
13         title = parent.title
14         if not title.startswith('Re:'):
15             title = 'Re: ' + parent.title
16
17         dc = getAdapter(parent, ICMFDublinCore)
18         formatter = self.request.locale.getDateTimeFormatter(
```

```
19              'medium')
20          body = '%s on %s wrote:\n' %(dc.creators[0],
21                                       formatter.format(dc.created))
22          body += '> ' + parent.body.replace('\n', '\n> ')
23
24          setUpWidgets(self, self.schema, IInputWidget
25                       initial={'title': title, 'body': body},
26                       names=self.fieldNames)
```

▷ Line 24–26: This is pretty much the original content of the `_setUpWidgets()` method, except for the `initial` argument, which carries the default values of the widgets.

Just register the form as

```
1 <addform
2     label="Reply to Message"
3     name="ReplyMessage"
4     schema="book.messageboard.interfaces.IMessage"
5     content_factory="book.messageboard.message.Message"
6     permission="book.messageboard.Add"
7     class=".views.ReplyMessage"
8     layer="board"/>
```

Once you restart Zope 3, you should be able to see the reply to Message screen.

This is how far I want to go with creating a nice, enduser-friendly interface. There is much more that could be done to make the package attractive, but it would not contain much content and would be boring to read about. Instead, I encourage the reader to decide him/herself about making further improvements. The exercises below should stimulate the reader about some possible improvements.

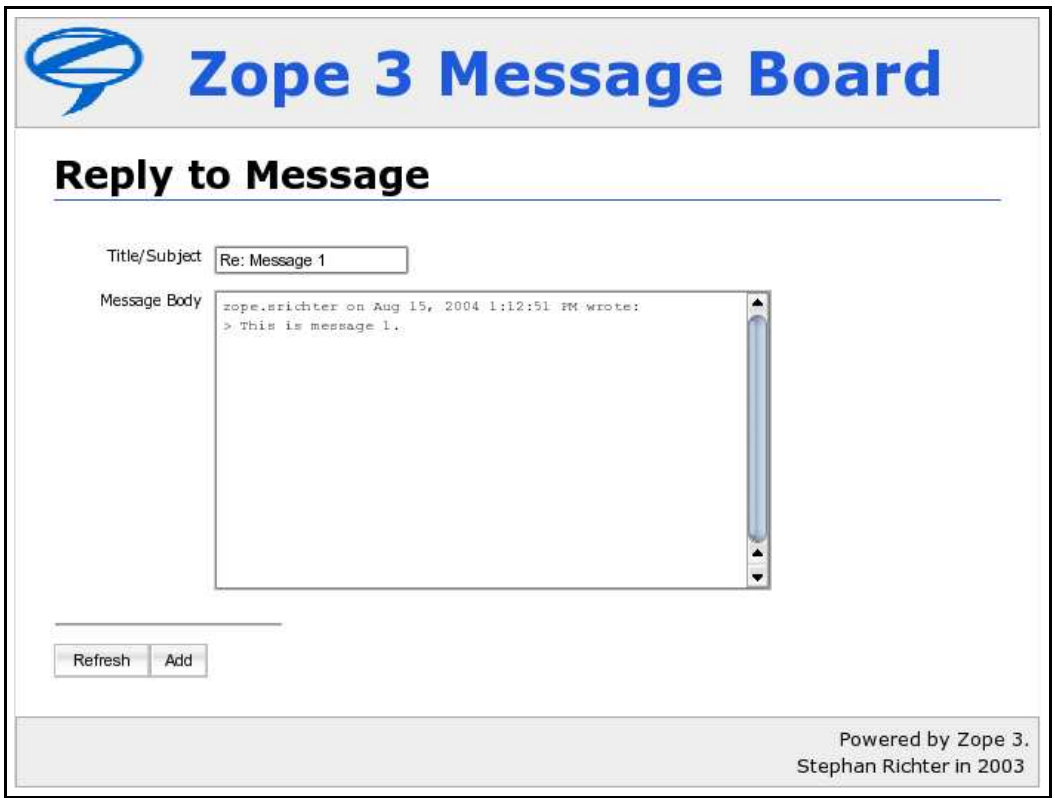


Figure 24.5: Reply to a message.

Exercises

1. If you don't like the generated message names, there is a way to change this. The names are generated by an adapter for `IWriteContainer` implementing `zope.app.container.interfaces.INameChooser`. Write you own adapter for `IMessage` and `IMessageBoard`

2. Currently the actions on the bottom of the page are implemented using simple links and the user does not know whether s/he will be able to access the view or not. Implement the actions in a way that they depend on the user's permissions. Note: Menus are extremely helpful for this, so I suggest using them.

3. The message board's "Posts" screen currently shows all published messages, no matter what. If you have a couple hundred messages, it can take quiet a while to load the screen. Therefore you should implement some batching that only shows at most 20 messages at a time.

4. There exists currently no way to add attachments to posts. Add a screen that allows you to add attachments to a message. You might also want to customize the "Create a new Post" and "Reply to Post" add forms to be able to add attachments.

5. The "Review Messages" for publication screen has really no option of declining messages. Therefore add some functionality that allows the refusal of messages. Once a message is refused, it should be either deleted or set to private state.

6. There is currently no way for users to mail-subscribe to messages. Implement this screen.

7. Utilize the Online Help and the existing screens to build a very user-friendly help for the message board.

PART V

Other Components

From the introduction chapters you know that content objects are not the only type of component you want to write. This section covers several of these various other components, such as utilities and resources.

Chapter 25: Building and Storing Annotations

Since it is desirable to leave an object as untouched as possible, we developed a mechanism for attaching meta data to an object without the object knowing about it.

Chapter 26: New Principal-Source Plug-Ins

Many people have very different requirements on the mechanism an authentication service should function and Zope 3 respects this need. There exists an Authentication Service that accepts plugins to provide principal data from external data sources.

Chapter 27: Principal Annotations

A common task is to append data to principals. Since principals are often imported from external data sources, they are not attribute annotatable. This chapter will make use of the Principal Annotation service to store additional data.

Chapter 28: Creating a new Browser Resource

This is a short chapter telling the reader how to implement a new file system based resource (a context independent view).

Chapter 29: Registries with Global Utilities

Utilties can be used to implement Zope-wide registries. Since registries are so very useful, this chapter is a must.

Chapter 30: Local Utilities

While we saw already how simple it is to write global utilities, there is some more work to do for local utilities, which will be introduced in this chapter.

Chapter 31: Vocabularies and Related Fields/Widgets

Vocabularies are a powerful extension to the schema framework and provide a generic method to create fields with variable and context dependent selection lists.

Chapter 32: Exception Views

For Zope 3, exceptions are simply objects, which can have views that make them representable on any output medium (usually the browser). Every exception has a standard view, but for a professional Web site you will need better screens.

CHAPTER 25

BUILDING AND STORING ANNOTATIONS

Difficulty

Sprinter

Skills

- Be familiar with the concept of annotations and how they are used. See the WebDAV Namespace chapter for a good example of how to *use* annotations.
- Be comfortable with the component architecture and ZCML.

Problem/Task

Currently, every object that comes with Zope 3 and can have some sort of annotation, uses attribute annotations. Attribute annotations store the annotation data directly in the objects. This implementation works fine as long as the object is persistent and is stored in the ZODB. But what if you have SQL-based objects, such as in relational-to-object mapping solutions? Storing annotations on the attribute of the object would certainly not work. In these scenarios it becomes necessary to implement a custom annotations implementation. This chapter will demonstrate how this can be done.

Solution

25.1 Introduction

Before we can dive into developing a new annotation adapter, it is important to understand the inner-workings. First, there exists an interface named `IAnnotatable`. By providing this interface, an object declares that it is possible to store annotations for itself.

However, `IAnnotable` is too general, since it does not specify how the annotation can be stored and should therefore never be provided directly. One should never assume that one method works for all possible objects.

Zope 3 comes by default with an `IAttributeAnnotatable` interface that allows you to store the annotations in the attribute `__annotations__` on the object itself. This works well for any object whose instances are stored in the ZODB.

As second part to the equation we have the `IAnnotations` interface, which provides a simple mapping API (i.e. dictionary-like) that allows you to look up annotation data using a unique key. This interface is commonly implemented as an adapter requiring `IAnnotatable` and providing `IAnnotations`. Thus we need to provide an implementation for `IAnnotations` to have our own annotations storage mechanism.

For `IAttributeAnnotable` we have an `AttributeAnnotations` adapter. Note that by definition `IAnnotations` extends `IAnnotable`, since an `IAnnotation` can always adapt to itself.

Another important aspect about annotations is the key (unique id) that is being used in the mapping. Since annotations may contain a large amount of data, it is important to choose keys in a way that they will always be unique. The simplest way to ensure this is to include the package name in the key. So for dublin core meta data, for example, instead of using "ZopeDublinCore" as the key one should use "zope.app.dublincore.ZopeDublinCore". Some people also use a URI-based namespace notation: `http://namespace.zope.org/dublincore/ZopeDublinCore/1.0`.

25.2 Implementing an Alternative Annotations Mechanism

So, let's say we cannot store annotations in an attribute on the object, because the object is volatile. Where could we store the annotation data then? One possibility would be to use RDBs, which makes sense, if your object's data is also stored in a relational database. A derivative of this solution would be to use files; but this is hard to get working with transactions.

We also have to bear in mind, that annotations are used by Zope to store meta data such as dublin core meta data or workflow data. It would be hard to store this

data in a RDB. So the ZODB is still a good place to to store the annotation data. One way would be to develop some sort of annotations container. But it would be even better, if the annotations could store their annotations to a nearby object that implements `IAttibuteAnnotable`.

25.3 Step I: Developing the Interfaces

We don't want to put the burden of holding other object's annotations on just any object that we can find. Instead, an object should declare that it is willing to keep annotations of other objects by implementing an interface named `IAnnotationKeeper`.

Objects that want their annotations to be stored in annotation keepers, need to implement `IKeeperAnnotable`.

By the way, for an object to find a keeper, it must also implement `ILocation`. So clearly, the keeper annotation design makes only sense, if you expect the object to be stored in a traversable path and is therefore not generally applicable either.

For writing the interfaces, you first need to start a new package in `ZOPE3/src/book` called `keeperannotations`. Add the following contents in `interfaces.py`:

```
1  from zope.interface import Interface
2  from zope.app.annotation.interfaces import IAnnotatable
3
4  class IAnnotationKeeper(Interface):
5      """Marker indicating that an object is willing to store other object's
6      annotations in its own annotations.
7
8      This interface makes only sense, if the object that implements this
9      interface also implements 'IAnnotatable' or any sub-class.
10     """
11
12 class IKeeperAnnotatable(IAnnotatable):
13     """Marker indicating that an object will store its annotations in an
14     object implementing IAnnotationKeeper.
15
16     This requires the object that provides this interface to also implement
17     ILocation.
18
19     This interface does not specify how the keeper may be found. This is up
20     to the adapter that uses this interface to provide 'IAnnotations'.
21     """
```

Both interfaces are just markers, since there is no direct duty involved by implementing these interfaces.

25.4 Step II: The KeeperAnnotations Adapter

Implementing the keeper annotations adapter is simply a matter of implementing the `IAnnotations` interface's mapping methods. The tricky part of the implementation

is to find the object that is the annotation keeper. This should be no problem as long as (1) the object has a location and (2) a keeper is available in the path of the object.

This first issue is important. When an object was just created, an `ObjectCreatedEvent` is sent out to which the dublin core mechanism listens. The dublin core mechanism then tries to set the creation and modification date on the object using annotations. This is of course a problem, since the object has no location at this point. In these cases we need some sort of temporary storage that can keep the annotation until the object has a location.

The other issue we safely ignore. The easiest would be to simply make the root folder an `IAnnotationKeeper`. This way the lookup for a keeper will never fail provided the object has a location. In the `__init__.py` file of your package, add the following adapter code:

```
1  from BTrees.OOBTree import OOBTree
2
3  from zope.interface import implements
4  from zope.proxy import removeAllProxies
5
6  from zope.app import zapi
7  from zope.app.annotation.interfaces import IAnnotations
8
9  from interfaces import IKeeperAnnotatable, IAnnotationKeeper
10
11 keeper_key = 'book.keeperannotation.KeeperAnnotations'
12
13 tmp = {}
14
15 class KeeperAnnotations(object):
16     """Store the annotations in a keeper.
17     """
18     implements(IAnnotations)
19     __used_for__ = IKeeperAnnotatable
20
21     def __init__(self, obj):
22         self.obj = obj
23         self.obj_key = removeAllProxies(obj)
24         self.keeper_annotations = None
25
26         # Annotations might be set when object has no context
27         if not hasattr(obj, '__parent__') or obj.__parent__ is None:
28             self.keeper_annotations = tmp
29             return
30
31         for parent in zapi.getParents(obj):
32             if IAnnotationKeeper.providedBy(parent):
33                 # We found the keeper, get the annotation that will store
34                 # the data.
35                 annotations = IAnnotations(parent)
36                 if not annotations.has_key(keeper_key):
37                     annotations[keeper_key] = OOBTree()
38                 self.keeper_annotations = annotations[keeper_key]
39
```

```
40          if self.keeper_annotations == None:
41              raise ValueError, 'No annotation keeper found.'
42
43          # There are some temporary stored annotations; add them to the keeper
44          if tmp.has_key(obj):
45              self.keeper_annotations[self.obj_key] = tmp[obj]
46              del tmp[obj]
47
48      def __getitem__(self, key):
49          """See zope.app.annotation.interfaces.IAnnotations"""
50          annotations = self.keeper_annotations.get(self.obj_key, {})
51          return annotations[key]
52
53      def __setitem__(self, key, value):
54          """See zope.app.annotation.interfaces.IAnnotations"""
55          if not self.keeper_annotations.has_key(self.obj_key):
56              self.keeper_annotations[self.obj_key] = OOBTree()
57          self.keeper_annotations[self.obj_key][key] = value
58
59      def get(self, key, default=None):
60          """See zope.app.annotation.interfaces.IAnnotations"""
61          try:
62              return self[key]
63          except KeyError:
64              return default
65
66      def __delitem__(self, key):
67          """See zope.app.annotation.interfaces.IAnnotations"""
68          del self.keeper_annotations[self.obj_key][key]
```

- Line 11: This string will be used in the keeper's annotations to store the other objects' annotations.
- Line 13: This is the temporary annotations variable, which holds annotations for objects that have not yet been located (i.e. have no parent and name).
- Line 23: Here we use the object itself as key in the annotations. This works well for persistent objects, but volatile content components would still not work. See exercise 1, which addresses this issue.

  Note: We could not use the path of the object yet either, since the object might not have been located yet.
- Line 27–29: In the case that no location has been assigned to the object yet, use the temporary storage.
- Line 31–38: Walk through all the parents and try to find the closest keeper. When a keeper is found, use it. Next, add a keeper b-tree, if none exists yet.
- Line 40–41: If no keeper was found, raise a `ValueError`. The condition should never be true, since this would cause a lot of application code to fail. So make sure, that a keeper can always be found.

- Line 44–46: If there are some temporary annotation entries, it is time to move them to the real keeper now and delete it from the temporary storage.
- Line 48–51: A straightforward implementation that simply looks for the key at the correct place. A default is passed in, since the annotation for the given object might not even exist yet.
- Line 53–57: First the code checks whether an annotation entry already exists for the object in the keeper. If not, then a new entry is added. Finally, the new annotation for the object is set.
- Line 59–64: This implementation reuses the `__getitem__()` method to avoid code duplication.
- Line 66–68: Delegate the deletion request to the correct entry.

This implementation is not meant to be used in production, but to serve as a simple demonstration. The exercises address the most obvious issues and ask the reader to fix them with some guidance.

25.5 Step III: Unit Testing

The problem with writing annotations is that they are used by other content objects. The correct way of writing the tests would be to develop dummy content objects. But in order to keep the testing code as small as possible, we are just going to depend on some of the existing content implementations, namely `Folder` and `File`. The most well-defined annotation is probably the `ZopeDublinCore`, so we will use it to declare and test some annotations.

We will use doc strings to write the tests. For clarity of this chapter, the doc string tests are not listed here. You can find the test code in the class doc string of `KeeperAnnotations`.

We now just have to write a tests module to execute the doc tests. Create a file called `tests.py` and add the following lines. There are a lot of imports, since we have to register a bunch of adapters for the testing code to work.

You can now execute the tests fromt the Zope 3 root directory using

```
python tests.py -vpu --dir src/book/keeperannotations
```

25.6 Step IV: Configuration of the `KeeperAnnotations` Component

To use the `KeeperAnnotations` adapter, we need to register it.

In `configure.zcml` add the following lines:

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

  <adapter
      for=".interfaces.IKeeperAnnotatable"
      provides="zope.app.annotation.interfaces.IAnnotations"
      factory=".KeeperAnnotations" />

</configure>
```

And that's it for the package. The annotations keeper adapter is set up an can be used by other components.

25.7 Step V: Writing Functional Tests and Configuration

Functional tests are very important for this code, since we do not know whether we thought of all exceptions until we test the keeper annotations code in a fully running Zope 3 environment. In a running Zope 3 setup we cannot use the `File` as test object anymore, since it already implements `IAttributeAnnotable`. Instead, we will use a `KeeperFile` class that inherits from `File` and register it seperately. In a new file named `ftests.py` add the new content type.

```
from zope.app.file import File

class KeeperFile(File):
    pass
```

We now need to register this new content type. We could add the registration directives to `configure.zcml` and be done, but since this content type is for testing only, it is better to create a separate configuration file and hook it up with the functional testing configuration directly.

We register the `KeeperFile` in almost identically the same way as the regular `File`. Additionally, the `KeeperAnnotations` adapter must be registered. For simplicity we make every `Folder` an `IAnnotationKeeper`.

Create a new file `keeper.zcml` and add the following lines.

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

  <content class="zope.app.folder.Folder">
    <implements
      interface=".interfaces.IAnnotationKeeper"
      />
  </content>

```

```
12   <content class=".ftests.KeeperFile">
13     <factory
14         id="KeeperFile"
15         title="Keeper File"
16         description="A Keeper File" />
17
18     <require
19         permission="zope.View"
20         interface="zope.app.filerepresentation.interfaces.IReadFile" />
21
22     <require
23         permission="zope.ManageContent"
24         interface="zope.app.filerepresentation.interfaces.IWriteFile"
25         set_schema="zope.app.filerepresentation.interfaces.IReadFile"
26         />
27
28     <implements
29       interface=".interfaces.IKeeperAnnotatable"
30       />
31   </content>
32
33   <browser:addMenuItem
34       class=".ftests.KeeperFile"
35       title="Keeper File"
36       permission="zope.ManageContent"
37       view="KeeperFile"
38       />
39
40   <browser:addform
41       schema="zope.app.file.interfaces.IFile"
42       label="Add a Keeper File"
43       content_factory=".ftests.KeeperFile"
44       name="AddKeeperFile"
45       permission="zope.ManageContent"
46       />
47
48 </configure>
```

If you do not understand these directives, read some of the content component chapters, as they explain them in much detail.

We now need to tell the functional configuration setup, that it should evaluate `keeper.zcml`. We do so by adding

```
1 <include file="src/book/keeperannotations/keeper.zcml" />
```

to `ZOPE3/ftesting.zcml`. This file is the main configuration file for functional tests. It is used in the same way that `site.zcml` is used for a real start up.

Now we should be configured and can concentrate on writing the test itself. The test will emulate pretty much the unit tests. Here is the setup, which should be added to `ftests.py`.

```
1 import time
2 import unittest
3
4 from zope.app.file import File
5 from zope.app.tests.functional import BrowserTestCase
```

```
6
7  from book.keeperannotations import keeper_key
8
9  class KeeperFile(File):
10     pass
11
12
13 class Test(BrowserTestCase):
14     """Funcional tests for Keeper Annotations.
15     """
16
17     def test_DC_Annotations(self):
18         # Create file
19         response = self.publish(
20             "/+/action.html?type_name=KeeperFile",
21             basic='mgr:mgrpw')
22
23         self.assertEqual(response.getStatus(), 302)
24         self.assertEqual(response.getHeader('Location'),
25                          'http://localhost/@@contents.html')
26
27         # Update the file's title
28         self.publish("/@@contents.html",
29                      basic='mgr:mgrpw',
30                      form={'retitle_id' : 'KeeperFile',
31                            'new_value' : u'File Title'})
32
33         root = self.getRootFolder()
34         file = root['KeeperFile']
35         ann = root.__annotations__[keeper_key][file]
36         dc_ann = ann['zope.app.dublincore.ZopeDublinCore']
37         self.assert_(dc_ann[u'Date.Created'][0] > u'2004-01-01T12:00:00')
38         self.assert_(dc_ann[u'Date.Created'][0] == dc_ann[u'Date.Modified'][0])
39         self.assertEqual(dc_ann[u'Title'][0], u'File Title')
40
41
42 def test_suite():
43     return unittest.TestSuite((
44         unittest.makeSuite(Test),
45         ))
46
47 if __name__=='__main__':
48     unittest.main(defaultTest='test_suite')
```

- Line 19–25: Add a keeper file to the root folder and make sure the correct HTTP response its provided.

- Line 28–31: Now change the title of the file. The title is provided by the Dublin Core, which in turn uses annotations to store its values.

- Line 33–40: Testing the correct behavior using HTML views would be too tedious, so we just grab the root folder directly and analyze the annotations for correct entries like we did it in the unit tests.

You can now run the tests as usual. You can also try the annotation keeper with the messageboard. Simply make the message board an annotation keeper for messages.

You can find the complete source code at `http://svn.zope.org/book/trunk/keeperannotations`.

Exercises

1. Using the object itself as key for the annotations works well for persistent objects, but fails for volatile ones. Develop an interface and adapter that will provide a unique id for a given object. While Zope X3 3.0.0 does not ship with a unique id generator, the trunk has a unique id utility, which can be used to create such ids for objects. You might want to use that utility.

2. Storing the annotations in a dictionary called `tmp` until the object has been placed somewhere is not the best solution. It would be better, if the temporary annotations would be stored in a file. Hint: You might want to use `pickle` to convert the Python data structures to a string and back.

3. Currently, if you only set annotations before the object is assigned a location, then the annotations will last forever in the temporary annotations dictionary. If you shut down the server, you loose the annotations (using the approach asked for in exercise 2 solves the issue). The better approach, however, would be to write an event subscriber that listens for events that notify the system that a location was assigned to an object. It could then move the data as response. Write such an event subscriber and hook it up to the existing code.

4. The existing implementation lacks the ability to handle moving and copying of objects. Why is that bad? When an object is moved, its keeper might change; therefore the object will effectively loose its annotations. The key, again, would be to implement an event subscriber that listens to move and copy events. When an object is moved and the keeper changes, the subscriber will copy the annotation data from the old keeper to the new one. Implement this!

CHAPTER 26

NEW PRINCIPAL-SOURCE PLUG-INS

Difficulty

Sprinter

Skills

- You should have a basic understanding of the Zope 3 component architecture.
- It is necessary to understand the purpose and differences between permissions, roles and principals.
- Basic knowledge about the Authentication Service. Optional.

Problem/Task

Many systems provide their own mechanisms for authentication. Examples include `/etc/passwd`, LDAP, NIS, Radius and relational databases. For a generic platform like Zope it is critically necessary to provide facilities to connect to these external authentication sources.

Zope 3 provides an advanced Authentication Service that provides an interface to integrate any external authentication source by simply developing a small plug-in, called a principal source. In this chapter we create such a plug-in and register it with the Authentication Service .

Solution

While one can become very fancy in implementing a feature-rich principal source implementation, we are concentrating on the most simple case here. The exercises point out many of the improvements that can be done during later development.

The goal of this chapter is to create a file-based principal source, so that it could read a `/etc/passwd`-like file (it will not actually be able to read `passwd` files, since we do not know whether everyone has the `crypt` module installed on her/his machine). The format of the file that we want to be able to read and parse is (users are separated by a newline character):

```
login:password:title:other_stuff
```

Let's now turn to the principal source. A component implementing `ILoginPasswordPrincipalS` (which extends `IPrincipalSource`) promises to provide three simple methods:

- `getPrincipal(id)` – This method gets a particular principal by its id, which is unique for this particular source. If no principal with the supplied id is found, a `NotFoundError` is raised.
- `getPrincipals(name)` – This method returns a list of principals whose login somehow contains the substring specified in `name`. If `name` is an empty string, all principals of this source are returned.
- `authenticate(login,password)` – This method is actually required by the `ILoginPasswordPrincipalSource` interface and provides authentication for the provided principal. There are other ways to implement authentication for these principals, but they add unnecessary complexity. `None` is returned, if no match is made.

The next step is to provide an implementation of `ILoginPasswordPrincipalSource` for password files. Create a new sub-package called `passwdauth` in the `book` package. Now the first step is to define the interfaces, as usual.

26.1 Step I: Defining the interface

"What interface do we need?", you might wonder. In order for a file-based principal source plug-in to provide principals, we need to know the file that contains the data; knowing about this file is certainly part of the API for this plug-in. So we want to create a specific interface that contains a filename. If we make this attribute a schema field, we can even use the interface/schema to create autogenerated add and edit forms.

In the `passwdauth` directory add an `interfaces.py` file and add the following contents:

```
from zope.schema import TextLine
from zope.app.i18n import ZopeMessageIDFactory as _

from zope.app.pluggableauth.interfaces import IPrincipalSource

```

```
6  class IFileBasedPrincipalSource(IPrincipalSource):
7      """Describes file-based principal sources."""
8
9      filename = TextLine(
10         title = _(u'File Name'),
11         description=_(u'File name of the data file.'),
12         default = u'/etc/passwd')
```

- ▷ Line 1: Here we have the usual imports of the TextLine field for the filename property.
- ▷ Line 2: This is the typical I18n boilerplate (not much though); all text strings wrapped by the underscore function will be internationalized, or in other terms localizable.
- ▷ Line 4: Our file-based principal source is still of type `IPrincipalSource`, so let's make it the base interface.
- ▷ Line 9–12: Typical internationalized text line field declaration, making `/etc/passwd` the default value (even though the product will not work with this file due to the `crypt` module issue). You might want to add a different default, also based on the operating system you are on.

26.2 Step II: Writing the tests

The next step is to write some unit tests that assure that the file parser does its job right. But first we need to develop a small data file with which we can test the plug-in with. Create a file called `passwd.sample` and add the following two principal entries:

```
1 foo1:bar1:Foo Bar 1
2 foo2:bar2:Foo Bar 2
```

Now we have a user with login `foo1` and one known as `foo2`, having `bar1` and `bar2` as passwords, respectively.

In the following test code we will only test the aforementioned three methods of the principal source. The file reading code is not separately checked, since it will be well tested through the other tests.

Create a `tests.py` file and add the code below.

```
1 import os
2 import unittest
3
4 from zope.exceptions import NotFoundError
5
6 from book import passwdauth
7
```

```
 8
 9 class PasswdPrincipalSourceTest(unittest.TestCase):
10
11     def setUp(self):
12         dir = os.path.dirname(passwdauth.__file__)
13         self.source = passwdauth.PasswdPrincipalSource(
14             os.path.join(dir, 'passwd.sample'))
15
16     def test_getPrincipal(self):
17         self.assertEqual(self.source.getPrincipal('\t\tfoo1').password, 'bar1')
18         self.assertEqual(self.source.getPrincipal('\t\tfoo2').password, 'bar2')
19         self.assertRaises(NotFoundError, self.source.getPrincipal, '\t\tfoo')
20
21     def test_getPrincipals(self):
22         self.assertEqual(len(self.source.getPrincipals('foo')), 2)
23         self.assertEqual(len(self.source.getPrincipals('')), 2)
24         self.assertEqual(len(self.source.getPrincipals('2')), 1)
25
26     def test_authenticate(self):
27         self.assertEqual(self.source.authenticate('foo1', 'bar1')._id, 'foo1')
28         self.assertEqual(self.source.authenticate('foo1', 'bar'), None)
29         self.assertEqual(self.source.authenticate('foo', 'bar1'), None)
30
31 def test_suite():
32     return unittest.makeSuite(PasswdPrincipalSourceTest)
33
34 if __name__=='__main__':
35     unittest.main(defaultTest='test_suite')
```

▷ Line 1, 12–14: The reason we imported `os` was to be able to get to the directory of the code as seen in line 12. Once we have the directory it is easy to build up the data file path and initialize the principal source (line 13–14).

▷ Line 16–19: Test the `getPrincipal(id)` method. The last test checks that the correct error is thrown in case of a failure. The full principal id is usually a tab-separated string of an earmark, the principal source name and the principal id. Since we do not have an earmark or a principal source name specified in a unit tests, these two values are empty and the full principal id has two tab characters at the beginning.

▷ Line 21–24: The test for `getPrincipals(name)` mainly tests that the resulting user list is correctly filtered based on the `name` parameter value.

▷ Line 26–29: The authentication test concentrates on checking that really only a valid login name and password pair receives a positive authentication by returning the principal object.

▷ Line 31–35: This is the usual test boiler plate.

You can later run the tests either using Zope's `test.py` test runner or by executing the script directly; the latter method requires the Python path to be set correctly to `ZOPE3/src`.

26.3 Step III: Implementing the plug-in

The implementation of the plug-in should be straightforward and bear no surprises. The tests already express all the necessary semantics. We only have not discussed the data structure of the principal itself yet. Here we can reuse the `SimplePrincipal`, which is a basic `IUser` implementation that contains all the data fields (`IUserSchemafied`) relevant to a principal: id, login (username), password, title and description.

Note that in Zope 3 the principal knows absolutely nothing about its roles, permissions or anything else about security. This information is handled by other components of the system and is subject to policy settings. Now we are ready to realize the principal source. In the `__init__.py` file of the `passwdauth` package we add the following implementation:

```
1  import os
2  from persistent import Persistent
3
4  from zope.exceptions import NotFoundError
5  from zope.interface import implements
6
7  from zope.app.container.contained import Contained
8  from zope.app.location import locate
9  from zope.app.pluggableauth import SimplePrincipal
10 from zope.app.pluggableauth.interfaces import IContainedPrincipalSource
11 from zope.app.pluggableauth.interfaces import ILoginPasswordPrincipalSource
12
13 from interfaces import IFileBasedPrincipalSource
14
15 class PasswdPrincipalSource(Contained, Persistent):
16     """A Principal Source for /etc/passwd-like files."""
17
18     implements(ILoginPasswordPrincipalSource, IFileBasedPrincipalSource,
19                IContainedPrincipalSource)
20
21     def __init__(self, filename=''):
22         self.filename = filename
23
24     def readPrincipals(self):
25         if not os.path.exists(self.filename):
26             return []
27         file = open(self.filename, 'r')
28         principals = []
29         for line in file.readlines():
30             if line.strip() != '':
31                 user_info = line.strip().split(':', 3)
32                 p = SimplePrincipal(*user_info)
```

```
33                locate(p, self, p._id)
34                p._id = p.login
35                principals.append(p)
36        return principals
37
38    def getPrincipal(self, id):
39        """See 'IPrincipalSource'."""
40        earmark, source_name, id = id.split('\t')
41        for p in self.readPrincipals():
42            if p._id == id:
43                return p
44        raise NotFoundError, id
45
46    def getPrincipals(self, name):
47        """See 'IPrincipalSource'."""
48        return filter(lambda p: p.login.find(name) != -1,
49                      self.readPrincipals())
50
51    def authenticate(self, login, password):
52        """See 'ILoginPasswordPrincipalSource'. """
53        for user in self.readPrincipals():
54            if user.login == login and user.password == password:
55                return user
```

▷ Line 2 & 14: Make sure the principal source object itself is persistent, so that it can be stored in the Authentication Service.

▷ Line 4: The `NotFoundError` is a Zope-specific exception, so we need to import it.

▷ Line 7 & 14: Since the principal source is stored inside an authentication service, we need to make it an `IContained` object.

▷ Line 8: The `locate()` method helps us assigning a location to objects, in this case principals. Since principals are contained by principal sources, we need to assign a parent and a name to them when they are created.

▷ Line 9: Here you can see where the `SimplePrincipal` is defined. There is really no need to implement our own version, even though it is a persistent class – we never add it to any object in the ZODB anyways.

▷ Line 10–13, & 18–19: Import the three principal source interfaces we promise to implement in our new principal source. The `IContainerPrincipalSource` makes sure that the principal source can only be added to a pluggable authentication service and nowhere else.

▷ Line 21–22: We need to make sure the `filename` attribute always exists; optionally it can even be passed to the constructor; we will make use of this fact in the autogenerated add form.

▷ Line 24–36: The `readPrincipals()` method does all the heavy lifting as it is responsible for reading and "parsing" the file. It contains all the logic for interpreting the file format. `readPrincipals()` is just a helper method and is therefore not defined in any interface.

- Line 25–26: In the first `if` statement the algorithm checks that the file really exists and return an empty list if it does not. This prohibits Zope from crashing if the file is not found, which is desirable in case you just made a simple typo and now you cannot access your Zope, because any authentication check will fail, since it passes through this code for every authentication call.
- Line 29: As mentioned before we assume that there is one line per user.
- Line 30: Let's ignore empty lines, they just cause headaches.
- Line 31–32: Another assumption is made; the entries in the file correspond directly to the arguments of the `SimplePrincipal` constructor, which is valid as long as the constructor signature of `SimplePrincipal` does not change.
- Line 33: Assign a location to the principal, so that we know where it came from.
- Line 34: The principal's `login` is generally different from its `id` field. Since we do not just want to support `/etc/passwd` files, we are not going to reuse the Unix user id, but simply use the login for its id.

▷ Line 38–44: This implementation of the `getPrincipal()` method reads all principals in and checks whether one with a matching id is found; if not, raise a `NotFoundError`. This is of course horribly inefficient and one should use caching – see Exercise 5 at the end of this chapter.

The principal id that is passed into this method argument really exists of three parts separated by a tab-character. The first part is the earmark (or unique id) of the authentication service, the second the name of the principal source and the third the id of the principal (line 38). However, we are only interested in the last part, which we use for comparison.

▷ Line 46–49: Again we simply use the `readPrincipals()` result to built up the list of matching principals.

▷ Line 51–55: The `authenticate()` method simply wades through all the users and tries to find a matching `login`/ `password` pair. When a match is found, the principal object is returned. Note that Python returns `None`, if no return value is specified, which is the case if no match was determined.

You should now run the unit tests to make sure that the implementation behaves as expected.

26.4 Step IV: Registering the Principal Source and Creating basic Views

We now have to register the `PasswdPrincipalSource` as content and create a basic add/edit form, since we need to allow the user to specify a data file. Create a configuration file (`configure.zcml`) and add the following directives:

```
1  <configure
2      xmlns="http://namespaces.zope.org/zope"
3      xmlns:browser="http://namespaces.zope.org/browser"
4      i18n_domain="demo_passwdauth">
5
6  <content class=".PasswdPrincipalSource">
7    <factory
8        id="zope.app.principalsources.PasswdPrincipalSource"
9        />
10   <allow interface=".interfaces.IFileBasedPrincipalSource"
11       />
12   <require
13       permission="zope.ManageContent"
14       set_schema=".interfaces.IFileBasedPrincipalSource"
15       />
16 </content>
17
18 <browser:addform
19     schema=".interfaces.IFileBasedPrincipalSource"
20     label="Add file-based Principal Source in /etc/passwd style"
21     content_factory=".PasswdPrincipalSource"
22     arguments="filename"
23     name="AddPasswdPrincipalSourceForm"
24     menu="add_principal_source" title="/etc/passwd Principal Source"
25     permission="zope.ManageContent"
26     />
27 <browser:editform
28     schema=".interfaces.IFileBasedPrincipalSource"
29     label="Edit file-based Principal Source"
30     name="edit.html"
31     menu="zmi_views" title="Edit"
32     permission="zope.ManageContent"
33     />
34 </configure>
```

- ▷ Line 6–16: Define the principal source as content, create a factory for it and make the appropriate security declarations for the interfaces. While the factory id (line 8) is usually the same as the Python object path, this is not the case here. However, this poses no problem, since the only requirement is that the id is globally unique.
- ▷ Line 18–26: Create a simple autogenerated add form. We also specify that the filename is the first and only argument for the constructor.
- ▷ Line 27–33: Matching to the add form, this is a simple edit form for the file name. Plain and simple is enough in this case.

One last step must be taken before the package can be tested: We need to incorporate the package into the system. Therefore add a file named `passwdauth-configure.zcml` into the `package-includes` directory having the following content:

```
1 <include package="book.passwdauth" />
```

Now (re)start your Zope server and try the new plug-in.

26.5 Step V: Taking it for a test run

After you have restarted Zope, take a Web browser and access `http://localhost:8080/`. From the contents screen go to the configuration by clicking on the `ManageSite` link. You are probably being prompted to login. Make sure you are logging in as a user having the `zope.Manager` role.

If you do not have an Authentication Service yet, then add this service by clicking on the `AddService` link. Select `AuthenticationService` and give it the name `auth_service`. The service will be automatically registered and activated for you. After this is done, you are left in the `Registration` view of the authentication service itself. In the `Add:` box on the left side, you should now see two entries, one of which is `/etc/passwdPrincipalSource`. Click on the new principal source and enter `passwd` as the name of the principal source.

In the next screen you are being asked to enter the path of the file. Darn, you might think, I do not have a file yet. But don't worry, we still have the file we used for the tests, so we can reuse it (and we know it works). So enter the following path, replacing `ZOPE3` with the path to your Zope 3 installation:

```
1 ZOPE3/src/book/passwdauth/passwd.sample
```

After submitting the form you end up in the `Contents` view of the Authentication Service again. Unfortunately, we have not added a screen yet, telling us whether the file exists and it successfully found users. I leave this exercise for the reader.

Before we can use the new principals, however, we have to assign roles to them. So go to `http://localhost:8080/@@contents.html`. In the top right corner you will see a `Grant` menu option. Click on it. In the next screen click on `Grantrolestoprincipals`. Now you should be convinced that the new principal source works, since "Foo Bar 1" and "Foo Bar 2" should appear in the principal's list. Select "Foo Bar 1" and all of the listed roles and submit the form by pressing `Filter`. In the next screen you simply select `Allow` for all available roles, which assigns them to this user. Store the changes by clicking `Apply`.

We are finally ready to test the principal! Open another browser and enter the following URL: `http://localhost:8080/@@contents.html`. You will be prompted for a login. Enter `foo1` as username and `bar1` as password and it should show

you the expected screen, meaning that the user was authenticated and the role `SiteManager` was appropriately assigned. You should also see `User:FooBar1` somewhere on near the top of the screen.

Exercises

1. The chapter's implementation did not concentrate much on providing feedback to the user. It would be nice to have a screen with an overview of all of the login names with their titles and other information. Implement such a screen and add it as the default tab for the principal source.

2. The current implementation requires the passwords to be plain text, which is of course a huge security risk. Implement a version of the plug-in that can handle plain text, crypt-based and SHA passwords. Implement a setting in the user interface that lets the user select one of the encryption types.

3. Implement a version of the plug-in that provides a write interface, i.e. you should be able to add, edit and delete principals. It would be best to implement the entire `IContainerPrincipalSource` interface for this, since you can then make use of existing Container code.

4. It is very limiting to require a special file format for the principal data. It would be useful to develop a feature that allows the user to specify the file format. Implement this feature and provide an intuitive user interface for this feature. (This is a tough one; feel free to make some assumptions to solve the problem.)

5. Reading in the user data for every single authentication call is quiet expensive, so it would be helpful to implement some caching features. This can be done in two ways: (1) Use the caching framework to implement a cached version of the source or (2) save the list of principals in a volatile attribute (i.e. `_v_principals`) and check for every call whether the file had been modified since the last time it was read.

CHAPTER 27

PRINCIPAL ANNOTATIONS

Difficulty

Sprinter

Skills

- You should understand the concept of annotations.
- Be familiar with adapters and how to register them.

Problem/Task

A common task is to append meta-data to principals. However, principals are often imported from external data sources, so that they are not attribute annotatable. Therefore a different solution is desirable. The Principal Annotation service was developed to always allow annotations for a principal. This chapter will show you how to use the Principal Annotation service to store additional data.

Solution

We now know that we want to store additional meta-data for the principal, but what do we want to store? To make it short, let's provide an E-mail address and an IRC nickname. Since we do not want to hand-code the HTML forms, we will describe the two meta-data elements by an interface as usual.

But before we can write the interface, create a new package named `principalinfo` in the `book` package. Do not forget to add the `__init__.py` file.

27.1 Step I: The Principal Information Interface

Add file called `interfaces.py` in the newly created package. Then place the following interface in it.

```
from zope.i18n import MessageIDFactory
from zope.interface import Interface
from zope.schema import TextLine

_ = MessageIDFactory('principalinfo')


class IPrincipalInformation(Interface):
    """This interface additional information about a principal."""

    email = TextLine(
        title=_("E-mail"),
        description=_("E-mail Address"),
        default=u"",
        required=False)

    ircNickname = TextLine(
        title=_("IRC Nickname"),
        description=_("IRC Nickname"),
        default=u"",
        required=False)
```

The interface is straight forward. the two data elements are simply two text lines. If you wish, you could write a special `EMail` field that also checks for valid E-mail addresses.

27.2 Step II: The Information Adapter

The next task is to provide an adapter that is able to adapt from `IPrincipal` to `IPrincipalInformation` using the principal annotation service to store the data. In a new module named `info.py` add the following adapter code.

```
from persistent.dict import PersistentDict
from zope.interface import implements
from zope.app import zapi

from interfaces import IPrincipalInformation

key = 'book.principalinfo.Information'

class PrincipalInformation(object):
    r"""Principal Information Adapter"""
    implements(IPrincipalInformation)

    def __init__(self, principal):
        annotationsvc = zapi.getService('PrincipalAnnotation')
        annotations = annotationsvc.getAnnotations(principal)
        if annotations.get(key) is None:
```

```
17              annotations[key] = PersistentDict()
18          self.info = annotations[key]
19
20      def __getattr__(self, name):
21          if name in IPrincipalInformation:
22              return self.info.get(name, None)
23          raise AttributeError, "'%s' not in interface." %name
24
25      def __setattr__(self, name, value):
26          if name in IPrincipalInformation:
27              self.info[name] = value
28          else:
29              super(PrincipalInformation, self).__setattr__(name, value)
```

▷ Line 7: The key is used to uniquely identify the annotation that is used by this adapter.

▷ Line 8: Get the principal annotation service. Note that this code assumes that such a service exists. If not, a `ComponentLookupError` is raised and the initialization of the adapter fails. Luckily, when the ZODB is first generated it adds a principal annotation service to the root site manager.

▷ Line 9: Retrieve the set of annotations for the principal that was passed in. Internally, the annotation service uses the principal's id to store the annotations. Therefore it is important that a principal always keep its id or when it is changed, the annotation must be moved.

▷ Line 10–11: If the key was not yet registered for the principal, then initialize a persistent dictionary, which we will use to store the values of the fields.

▷ Line 12: The persistent data dictionary is set to be available as `info`.

▷ Line 14–17: If the name of the attribute we are trying to get is in the `IPrincipalInformation` interface, then retrieve the value from the `info` dictionary. If the name does not corresponds to a field in the interface, then raise an attribute error. Note that `__getattr__` is only called after the normal attribute lookup fails.

▷ Line 19–23: Similar to the previous method, if the name corresponds to a field in the `IPrincipalInformation` interface, then store the value in the data dictionary. If not, then use the original `__getattr__()` method to store the value.

This was not that hard, was it?

27.3 Step III: Registering the Components

Now that we have an adapter, we need to register it as such. Also, we want to create an edit form that allows us to edit the values.

```
1  <configure
2      xmlns="http://namespaces.zope.org/zope"
3      xmlns:browser="http://namespaces.zope.org/browser"
4      i18n_domain="principalinfo">
5
6    <adapter
7        factory=".info.PrincipalInformation"
8        provides=".interfaces.IPrincipalInformation"
9        for="zope.app.security.interfaces.IPrincipal"
10       permission="zope.ManageServices"
11       />
12
13   <browser:editform
14       name="userInfo.html"
15       schema=".interfaces.IPrincipalInformation"
16       for="zope.app.security.interfaces.IPrincipal"
17       label="Change User Information"
18       permission="zope.ManageServices"
19       menu="zmi_views" title="User Info" />
20
21 </configure>
```

▷ Line 6–11: The adapter is registered for all objects implementing `IPrincipal`. The entire `IPrincipalInformation` schema is available under the `zope.ManageServices` permission, which might not be desirable, but is sufficient for this example. For a real project, you would probably give the accessor a less strict permission than the attribute mutator. This can be done with a `zope:class` directive containing `zope:require` directives.

▷ Line 13–19: This edit form is registered for `IPrincipal` components, so that it will be available as a view for all principals. However, the schema that is being edited in `IPrincipalInformation`. The edit form will automatically lookup the adapter from `IPrincipal` to `IPrincipalInformation`.

You need to register the configuration with the Zope 3 framework by adding a file named `principalinfo-configure.zcml` to `package-includes` having the following one line directive.

```
1 <include package="book.principalinfo" />
```

You can now restart Zope 3, and the view should be available.

27.4 Step IV: Testing the Adapter

Before I show you how to use the Web interface to test the view, let's first write a test for the adapter to ensure the correct functioning. The most difficult part about the unit tests here is actually setting up the environment, such as defining and registering a principal annotation service. We will implement the test as a doctest in the `PrincipalInformation`'s doc string.

Let's first setup the environment.

```
1 >>> from zope.app.tests import setup
2 >>> from zope.app.principalannotation.interfaces import \
3 ...      IPrincipalAnnotationService
4 >>> from zope.app.principalannotation import PrincipalAnnotationService
5
6 >>> site = setup.placefulSetUp(site=True)
7 >>> sm = zapi.getGlobalServices()
8 >>> sm.defineService('PrincipalAnnotation',
9 ...                  IPrincipalAnnotationService)
10 >>> svc = setup.addService(site.getSiteManager(), 'PrincipalAnnotation',
11 ...                        PrincipalAnnotationService())
```

▷ Line 1: The `setup` module contains some extremely helpful convenience functions.

▷ Line 2–3: Import the interface that a principal annotation service must provide.

▷ Line 4: Import the implementation of the service.

▷ Line 6: Create a placeful setup, making the root folder a site, which is returned.

▷ Line 7–9: A new service type can only be defined via the global service manager, so get it first. Then define the service type by name and interface.

▷ Line 10–11: Add a principal annotation service to the site of the root folder.

Now that the service is setup, we need a principal to use the adapter on. We could use an existing principal implementation, but every that the principal annotation service needs from the principal is the `id`, which we can easily provide via a stub implementation.

```
1 >>> class Principal(object):
2 ...     id = 'user1'
3 >>> principal = Principal()
```

Now create the principal information adapter:

```
1 >>> info = PrincipalInformation(principal)
```

Before we give the fields any values, they should default to `None`. Any field not listed in the information interface should cause an `AttributeError`.

```
>>> info.email is None
True
>>> info.ircNickname is None
True
>>> info.phone
Traceback (most recent call last):
...
AttributeError: 'phone' not in interface.
```

Next we try to set a value for the email and make sure that it is even available if we reinstantiate the adapter.

```
>>> info.email = 'foo@bar.com'
>>> info.email
'foo@bar.com'

>>> info = PrincipalInformation(principal)
>>> info.email
'foo@bar.com'
```

Finally, let's make sure that the data is really stored in the service.

```
>>> svc.annotations['user1']['book.principalinfo.Information']['email']
'foo@bar.com'
```

Be careful to clean up after yourself.

```
>>> setup.placefulTearDown()
```

To make the tests runnable via the test runner, add the following test setup code to `tests.py`.

```
import unittest
from zope.testing.doctestunit import DocTestSuite

def test_suite():
    return DocTestSuite('book.principalinfo.info')

if __name__ == '__main__':
      unittest.main(defaultTest='test_suite')
```

Make sure that the test passes, before you proceed.

27.5 Step V: Playing with the new Feature

Now that the tests pass and the components are configured let's see the edit form. Restart Zope 3. Once restarted, go to `http://localhost:8080/++etc++site/default/manage` and click on "Authentication Service" in the "Add:" box. Once the authentication service is added, go to its "Contents" tab. Click on "Add Principal Source" in the "Add:" box and call it "btree", since it is a b-tree based persistent source. Enter the source's management screen and add a principal with any values. Once you enter the principal, you will see that a tab named "User Info" is available,

Figure 27.1: The principal's "User Info" screen.

which will provide you with the edit form created in this chapter. You can now go there and add the E-mail and IRC nickname of the principal.

Exercises

1. Currently the interface that is used to provide the additional user information is hard-coded. It would be nice, if the user could choose the interface s/he wishes to append as user information. Generalize the implementation, so that the user is asked to input the desired data interface as well.

CHAPTER 28

CREATING A NEW BROWSER RESOURCE

Difficulty

Newcomer

Skills

- Some basic ZCML knowledge is required.
- You should be familiar with the component architecture, especially presentation components and views.

Problem/Task

Certain presentation, like images and style sheets are not associated with any other component, so that one cannot create a view. To solve the problem resources were developed, which are presentation components that do not require any context. This mini-chapter will demonstrate how resources are created and registered with Zope 3.

Solution

The first goal is to register a simple plain-text file called `resource.txt` as a browser resource. The first step is to create this file anywhere you wish on the filesystem, and adding the following content:

```
1 Hello, I am a Zope 3 Resource Component!
```

Now just register the resource in a ZCML configuration file using the `browser:resource` directive:

```
1 <browser:resource
2     name="resource.txt"
3     file="resource.txt"
4     layer="default" />
```

▷ Line 2: This is the name under which the resource will be known in Zope.

▷ Line 3: The `file` attribute specifies the path to the resource on the filessytem. The current working directory (' .') is always the directory the configuration file is located. So in the example above, the file `resource.txt` is located in the same folder as the configuration file is.

▷ Line 4: The optional `layer` attribute specifies the layer the resource is added to. By default, the `default` layer is selected.

Once you hook up the configuration file to the main configuration path and restart Zope 3, you should be able to access the resource now via a Browser using `http://localhost:8080/@@/resource.txt`. The `@@/` in the URL tells the traversal mechanism that the following object is a resource.

If you have an image resource, you might want to use different configuration. Create a simple image called `img.png` and register it as follows:

```
1 <browser:resource
2     name="img.png"
3     image="img.png"
4     permission="zope.ManageContent" />
```

▷ Line 3: As you can see, instead of the `file` attribute we use the `image` one. Internally this will create an `Image` object, which is able to detect the content type and returns it correctly. There is a third possible attribute named `template`. If specified, a Page Template that is executed when the resource is called.

Note that only one of `file`, `image`, or `template` attributes can be specified inside a resource directive.

▷ Line 4: A final optional attribute is the "permission" one must have to view the resource. To demonstrate the security, I set the permission required for viewing the image to `zope.ManageContent`, so that you must log in as an administrator/-manager to be able to view it. The default of the attribute is `zope.Public` so that everyone can see the resource.

If you have many resource files to register, it can be very tedious to write a single directive for every resource. For this purpose the `resourceDirectory` is

provided, with which you can simply declare an entire directory, including its content as resources. Thereby the filenames of the files are reused as the names for the resource available. Assuming you put your two previous resources in a directory called `resource`, then you can use the following:

```
<browser:resourceDirectory
    name="resources"
    directory="../resource" />
```

The image will then be publically available under the URL: `http://localhost:8080/@@/resources/img.png`

The `DirectoryResource` object uses a simple resource type recognition. It looks at the filename extensions to discover the type. For page templates, currently the extensions “pt”, “zpt” and “html” are registered and for an image “gif”, “png” and “jpg”. All other extensions are converted to file resources. Note that it is not necessary to have a list of all image types, since only Browser-displayable images must be recognized.

CHAPTER 29

REGISTRIES WITH GLOBAL UTILITIES

Difficulty

Contributer

Skills

- Familiarity with the Component Architecture is required.
- Be comfortable with ZCML.
- You should know the message board example, since we are going to use it in the final step.

Problem/Task

We need registries all the time. It is a very common pattern. In fact, the Component Architecture itself depends heavily on registered component lookups to provide functionality. These registries, especially the utility service, can be used to provide application-level registries as well.

Solution

29.1 Introduction

The goal of this chapter is to develop a mechanism that provides image-representations of text-based smileys. For example, the text ":-)" should be converted to  . To

further complicate the problem, it is undesirable to just store smileys without any order. Many applications support multiple themes of smileys, so we want to do that as well. For example, the message board administrator should be able to choose the desired smiley theme for his message board.

Based on the requirements above, we want a registry of smiley themes that the user can choose from. Therefore, we will develop the theme as a utility and register it with a specific interface, `ISmileyTheme`. Thus, the entire utility service acts as a huge registry, but we can easily simulate sub-registries by specifying a particular interface for a utility. We can then ask the Zope component API to return a list of all utilities providing `ISmileyTheme` or simply return a single `ISmileyTheme` having a specific name.

Let's now take a look on how to handle the smileys themselves. Inside the theme, we simply need a mapping (dictionary) from the text representation to the image. However, should we really store the image data. In fact, it would better to declare the image itself as a resource and only store the URL, so that we can (a) support external links (not done in this chapter) and (b) do not have to worry about publishing the images.

The code will be rounded off by implementing a couple new ZCML directives to make the registration of new smiley themes as easy as possible, so that a message board editor can easily upload his/her favorite theme and use it. We will actually add a final step to the message board example incorporating smiley themes at the end of the chapter.

To allow the smiley theme utility to be distributed independently of the message board application, develop its code in a new package called `smileyutility`, which you should place into `ZOPE3/src/book`. Don't forget to add an `__init__.py` file to the directory.

29.2 Step I: Defining the Interfaces

Before we start coding away, we need to spend some time thinking about the API that we want to expose. In the next chapter we develop a local/placeful equivalent of the utility, so our base interface, `ISmileyTheme`, should be general enough to support both implementations and not include any implementation-specific methods. We will then derive another interface, `IGlobalSmileyTheme`, from the general one that will specify methods to manage smileys for the global implementation. Note: The utility will still be registered as a `ISmileyTheme`.

```
1 from zope.interface import Interface
2
3 class ISmileyTheme(Interface):
4     """A theme is a collection of smileys having a stylistic theme.
```

```
5
6       Themes are intended to be implemented as named utilities, which will be
7       available via a local smiley service.
8       """
9
10      def getSmiley(text, request):
11          """Returns a smiley for the given text and theme.
12
13          If no smiley was found, a ComponentLookupError should be raised.
14          """
15
16      def querySmiley(text, request, default=None):
17          """Returns a smiley for the given text and theme.
18
19          If no smiley was found, the default value is returned.
20          """
21
22      def getSmileysMapping(request):
23          """Return a mapping of text to URL.
24
25          This is incredibly useful when actually attempting to substitute the
26          smiley texts with a URL.
27          """
28
29
30  class IGlobalSmileyTheme(ISmileyTheme):
31      """A global smiley theme that also allows management of smileys."""
32
33      def provideSmiley(text, smiley_path):
34          """Provide a smiley for the utility."""
```

You might think that this interface seems a bit wordy, but in such widely available components it is extremely important to document the specific semantics of each method and the documentation should leave no question or corner case uncovered. Many people will depend on the correctness of the API.

- ▷ Line 1 & 3: Notice that a utility does not have to inherit any special interfaces. Until we declare a utility to be a utility, it is just a general component.
- ▷ Line 10–14: Retrieve a smiley given its text representation. Note that we need the request, since the URL could be built on the fly and we need all the request information to generate an appropriate URL.
- ▷ Line 16–20: Similar to the get-method, except that it returns default instead of raising an error, if the smiley was not found.
- ▷ Line 22–25: Interestingly enough, I did not have this method in my original design, but noticed that the service would be unusable without it. By returning a complete list of text-to-URL mappings, the application using this utility can simply do a search and replace of all smiley occurrences.

  In the beginning I envisioned a method that was taking a string as argument and returns a string with all the smiley occurrences being replaced by image tags.

But this would have been rather limiting, since the utility would need to guess the usage of the URL; not everyone wants to generate HTML necessarily. This implementation does not carry this restriction, since it makes no assumption on how the URLs will be used.

▷ Line 33–34: As an extension to the `ISmileyTheme` interface, this method adds a new smiley to the theme. The `smiley_path` will be expected to be a relative path to a resource, something like `++resource++plain__smile.png`. Note that the path must be unique, across all themes, so it is a good idea to encode the theme name into it by convention. But let's not introduce that restriction in the theme.

Now that we have all interfaces defined, let's look at the implementation, which should be straightforward.

29.3 Step II: Implementing the Utility

The global theme will use simple dictionary that maps a text representation to the path of a smiley. When smileys are requested this path is converted to a URL and returned. The only tricky part of the utility will be to obtain the root URL, since the utility does not know anything about locations.

However, there is an fast solution. We create a containment root component stub that implements the `IContainmentRoot` interface, for which the traversal mechanism is looking for while generating a path or URL. So here is what I did for obtaining the root URL:

```
1 from zope.app import zapi
2 from zope.app.traversing.interfaces import IContainmentRoot
3 from zope.interface import implements
4
5 class Root:
6     implements(IContainmentRoot)
7
8 def getRootURL(request):
9     return str(zapi.getView(Root(), 'absolute_url', request))
```

▷ Line 8–9: Return the root URL for a given request. The reason we need this request object is that it might contain information about the server name and port, additional namespaces like skin declarations or virtual hosting information.

The `absolute_url` view is defined for all objects and returns the URL the object is reachable at, given that it has enough context information.

Now we have all the pieces to implement the utility. I just used `pyskel.py` to create the skeleton and then filled it. Place the following and the `getRootURL()` code in a file called `globaltheme.py`:

```
1  om zope.component.exceptions import ComponentLookupError
2  om interfaces import IGlobalSmileyTheme
3
4  ass GlobalSmileyTheme(object):
5    """A filesystem based smiley theme."""
6    implements(IGlobalSmileyTheme)
7
8    def __init__(self):
9        self.__smileys = {}
10
11   def getSmiley(self, text, request):
12       "See book.smileyutility.interfaces.ISmileyTheme"
13       smiley = self.querySmiley(text, request)
14       if smiley is None:
15           raise ComponentLookupError, 'Smiley not found.'
16       return smiley
17
18   def querySmiley(self, text, request, default=None):
19       "See book.smileyutility.interfaces.ISmileyTheme"
20       if self.__smileys.get(text) is None:
21           return default
22       return getRootURL(request) + '/' + self.__smileys[text]
23
24   def getSmileysMapping(self, request):
25       "See book.smileyutility.interfaces.ISmileyTheme"
26       smileys = self.__smileys.copy()
27       root_url = getRootURL(request)
28       for name, smiley in smileys.items():
29           smileys[name] = root_url + '/' + smiley
30       return smileys
31
32   def provideSmiley(self, text, smiley_path):
33       "See book.smileyutility.interfaces.IGlobalSmileyTheme"
34       self.__smileys[text] = smiley_path
```

- Line 8–9: Initialize the registry, which is a simple dictionary. Note that I want this registry to be totally private to this class and noone else should be able to reach it.
- Line 11–16: This method does not do much, since we turn over all the responsibility to the next method. All we do is complain with a `ComponentLookupError` if there was no result (i.e. `None` was returned).
- Line 18–22: First, if the theme does not contain the requested smiley, then simply return the default value. Now that we know that there is a smiley available, construct the URL by appending the smiley path to the URL root.
- Line 36–46: We make a copy of all the smiley map. If the theme does not exist, an empty dictionary is created. In line 43–44 we update every smiley path with a smiley URL.
- Line 48–51: The smiley path is simply added with the text being the key of the mapping.

Our utility is now complete. However, we have not created a way to declare a default theme. To make life simple, the default theme is simply available under the name "default".

```
from interfaces import ISmileyTheme

def declareDefaultSmileyTheme(name):
    """Declare the default smiley theme."""
    utilities = zapi.getService(zapi.servicenames.Utilities)
    theme = zapi.getUtility(ISmileyTheme, name)
    # register the utility simply without a name
    utilities.provideUtility(ISmileyTheme, theme, 'default')
```

In the code above we simply look up the utility by its original name and then register it again using the name "default". By the way, this is totally legal and of practiced. One Utility instance can be registered multiple times using different interfaces and/or names.

Now, let's test our new utility.

29.4 Step III: Writing Tests

Writing tests for global utilities is usually fairly simple too, since you usually do not have to start up the component architecture. In this case, however, we have to do this, since we are looking up a view when asking for the root URL. We also have to register this view (`absolute_url`) in the first place, so it can be found later. In the `tests` package I created a `test_doc.py` and inserted the set up and tear down code there:

```
import unittest

from zope.interface import Interface
from zope.testing.doctestunit import DocTestSuite

from zope.app.tests import ztapi, placelesssetup

class AbsoluteURL:
    def __init__(self, context, request):
        pass
    def __str__(self):
        return ''

def setUp():
    placelesssetup.setUp()
    ztapi.browserView(Interface, 'absolute_url', AbsoluteURL)


def test_suite():
    return unittest.TestSuite((
        DocTestSuite('book.smileyutility.globaltheme',
                     setUp=setUp, tearDown=placelesssetup.tearDown),
        ))
```

```
24
25 if __name__ == '__main__':
26     unittest.main(defaultTest='test_suite')
```

▷ Line 8–12: This is a stub implementation of the absolute URL. We simply return nothing as root of the url.

▷ Line 14–16: We have seen a placeless unittest setup before; `placelesssetup.setUp()` brings up the basic component architecture and clears all the registries from possible entries. Line 16 then registers our stub-implementation of `AbsoluteURL` as a view.

▷ Line 21–22: Here we create a doctest suite using the custom setup function.

Now we just have to write the tests. In the docstring of the `GlobalSmileyTheme` class add the following doctest code:

```
1 Let's make sure that the global theme implementation actually fulfills the
2 `ISmileyTheme` API.
3
4 >>> from zope.interface.verify import verifyClass
5 >>> verifyClass(IGlobalSmileyTheme, GlobalSmileyTheme)
6 True
7
8 Initialize the theme and add a couple of smileys.
9
10 >>> theme = GlobalSmileyTheme()
11 >>> theme.provideSmiley(':-)', '++resource++plain__smile.png')
12 >>> theme.provideSmiley(';-)', '++resource++plain__wink.png')
13
14 Let's try to get a smiley out of the registry.
15
16 >>> from zope.publisher.browser import TestRequest
17
18 >>> theme.getSmiley(':-)', TestRequest())
19 '/++resource++plain__smile.png'
20 >>> theme.getSmiley(':-(', TestRequest())
21 Traceback (most recent call last):
22 ...
23 ComponentLookupError: 'Smiley not found.'
24 >>> theme.querySmiley(';-)', TestRequest())
25 '/++resource++plain__wink.png'
26 >>> theme.querySmiley(';-(', TestRequest()) is None
27 True
28
29 And finally we'd like to get a dictionary of all smileys.
30
31 >>> map = theme.getSmileysMapping(TestRequest())
32 >>> len(map)
33 2
34 >>> map[':-)']
35 '/++resource++plain__smile.png'
36 >>> map[';-)']
37 '/++resource++plain__wink.png'
```

▷ Line 4–6: It is always good to ensure that the interface was correctly implemented.

▷ Line 8–12: Test the `provideSmiley()` method.

▷ Line 14–27: Test the simple smiley accessor methods of the utility. Note how nicely doctests also handle exceptions.

▷ Line 29–37: Make sure that the `getSmileyMapping()` method gives the right output. Note that dictionaries cannot be directly tested in doctests, since its representation depends on the computer architecture, since the item order is arbitrary.

Run the tests and make sure that they all pass.

29.5 Step IV: Providing a user-friendly UI

While the current API is functional, it is not very practical to the developer, since s/he first needs to look up the theme using the component architecture's utility API and only then can make use the of the smiley theme features. It would be much nicer, if we would only need a smiley-theme-related API to work with. Thus we create some convenience functions in the package's `__init__.py` file:

```
1  from zope.app import zapi
2
3  from interfaces import ISmileyTheme
4
5  def getSmiley(text, request, theme='default'):
6      theme = zapi.getUtility(ISmileyTheme, theme)
7      return theme.getSmiley(text, request)
8
9  def querySmiley(text, request, theme='default', default=None):
10     theme = zapi.queryUtility(ISmileyTheme, theme)
11     if theme is None:
12         return default
13     return theme.querySmiley(text, request, default)
14
15 def getSmileyThemes():
16     return [name for name, util in zapi.getUtilitiesFor(ISmileyTheme)
17             if name != 'default']
18
19 def getSmileysMapping(request, theme='default'):
20     theme = zapi.getUtility(ISmileyTheme, theme)
21     return theme.getSmileysMapping(request)
```

The functions integrate the theme utility more tightly in the API.

▷ Line 15–17: Return a list of names of all available themes, excluding the "default" one.

The tests for these functions are very similar to the ones of the theme utility, so I am not going to include them in the text. As always, you can find the complete code including methods in the code repository (`http://svn.zope.org/book/smileyutility`).

29.6 Step V: Implement ZCML Directives

You might have already wondered, how this utility can be useful, if it does not even deal with the smiley images. This functionality is reserved for the configuration. When a smiley registration is made, the directive will receive a path to an image, but does not just register it with the smiley theme. Instead, it first creates a resource for the image and then passes the resource's relative path to the smiley theme.

In case you have not written a ZCML directive yet, there are three steps: creating the directive schema, implementing the directive handlers and writing the meta-ZCML configuration. They are represented by the next three sections (a) through (c).

But first we need to decide what directives we want to create. The first one, `smiley:theme`, defines a new theme and allows a sub-directive, `smiley:smiley`, that registers new smileys for this theme. A second directive, `smiley:smiley`, allows you to register a single smiley for an existing theme, so that other packages can add additional smileys to a theme. The third and final directive, `smiley:defaultTheme`, let's you specify the theme that will be known as the default one. The specified theme must exist already.

29.6.1 (a) Declaring the directive schemas

Each ZCML directive is represented by a schema, which defines the type of content for each element/directive attribute. Each field is also responsible for knowing how to convert the attribute value into something that is useful. All the usual schema fields are available. Additionally there are some specific configuration fields that can also be used. They are listed in the "Intorduction to the Zope Configuration Markup Language (ZCML)" chapter.

So now that we know what we can use, let's define the schemas. By convention they are placed in a file called `metadirectives.py`:

```
1 from zope.interface import Interface
2 from zope.configuration.fields import Path
3 from zope.schema import TextLine
4
5 class IThemeDirective(Interface):
6     """Define a new theme."""
7
8     name = TextLine(
```

```
9         title=u"Theme Name",
10        description=u"The name of the theme.",
11        default=None,
12        required=False)
13
14 class ISmileySubdirective(Interface):
15     """This directive adds a new smiley using the theme information of the
16     complex smileys directive."""
17
18     text = TextLine(
19         title=u"Smiley Text",
20         description=u"The text that represents the smiley, i.e. ':-)'",
21         required=True)
22
23     file = Path(
24         title=u"Image file",
25         description=u"Path to the image that represents the smiley.",
26         required=True)
27
28 class ISmileyDirective(ISmileySubdirective):
29     """This is a standalone directive registering a smiley for a certain
30     theme."""
31
32     theme = TextLine(
33         title=u"Theme",
34         description=u"The theme the smiley belongs to.",
35         default=None,
36         required=False)
37
38 class IDefaultThemeDirective(IThemeDirective):
39     """Specify the default theme."""
```

- Line 5–12: The `theme` directive only requires a "name" attribute that gives the theme its name.
- Line 13–25: Every smiley is identified by its text representation and the image file. (The theme is already specified in the sub-directive.)
- Line 27–35: This is the single directive that specifies all information at once. We simply reuse the previously defined `smiley` sub-directive interface and specify the theme.
- Line 37–38: The default theme directive is simple, because it just takes a theme name.

29.6.2 (b) Implement ZCML directive handlers

Next we implement the directive handlers themselves, which is the real fun part, since it actually represents some important part of the package's logic. This code goes by convention into `metaconfigure.py`:

```
1  import os
2
3  from zope.app import zapi
4  from zope.app.component.metaconfigure import utility
5  from zope.app.publisher.browser.resourcemeta import resource
6
7  from interfaces import ISmileyTheme
8  from globaltheme import GlobalSmileyTheme, declareDefaultSmileyTheme
9
10 __registered_resources = []
11
12 def registerSmiley(text, path, theme):
13     theme = zapi.queryUtility(ISmileyTheme, theme)
14     theme.provideSmiley(text, path)
15
16 class theme(object):
17
18     def __init__(self, _context, name):
19         self.name = name
20         utility(_context, ISmileyTheme,
21                 factory=GlobalSmileyTheme, name=name)
22
23     def smiley(self, _context, text, file):
24         return smiley(_context, text, file, self.name)
25
26     def __call__(self):
27         return
28
29 def smiley(_context, text, file, theme):
30     name = theme + '__' + os.path.split(file)[1]
31     path = '/++resource++' + name
32
33     if name not in __registered_resources:
34         resource(_context, name, image=file)
35         __registered_resources.append(name)
36
37     _context.action(
38         discriminator = ('smiley', theme, text),
39         callable = registerSmiley,
40         args = (text, path, theme),
41         )
42
43 def defaultTheme(_context, name=None):
44     _context.action(
45         discriminator = ('smiley', 'defaultTheme',),
46         callable = declareDefaultSmileyTheme,
47         args = (name,),
48         )
```

▷ Line 10: We want to keep track of all resources that we have already added, so that we do not register any resource twice, which would raise a component error.

▷ Line 12–14: Actually sticking in the smileys into the theme must be delayed till the configuration actions are executed. This method will be the smiley registration callable that is called when the smiley registration action is executed.

▷ Line 16–27: Since `theme` is a complex directive (it can contain other directives inside), it is implemented as a class. The parameters of the constructor resemble the arguments of the XML element, except for `_context`, which is always passed in as first argument and represents the configuration context.

Each sub-directive (in our case `smiley`) is a method of the class taking the element attributes as parameters. In this implementation we forward the configuration request to the main `smiley` directive; there is no need to implement the same code twice.

Every complex directive class must be callable (i.e. implement `__call__()`). This method is called when the closing element is parsed. Usually all of the configuration action is happening here, but not in our case.

▷ Line 29–41: The first task is to separate the filename from the file path and construct a unique name and path for the smiley. On line 33–35 we register the resource. We do that only, if we have not registered it before, which can happen if there are two text representations for a single smiley image, like ":)" and ":-)". On line 37–41 we then tell the configuration system it should add the smiley to the theme. Note that these actions are not executed at this time, since the configuration mechanism must first resolve possible overrides and conflict errors.

▷ Line 43–48: This is a simple handler for the simple `defaultTheme` directive. It calls our previously developed `declareDefaultSmileyTheme()` function and that's it.

29.6.3 (c) Writing the meta-ZCML directives

Now that we have completed the Python-side of things, let's register the new ZCML directives using the `meta` namespace in ZCML. By convention the ZCML directives are placed into a file named `metal.zcml`:

```
1  <configure xmlns:meta="http://namespaces.zope.org/meta">
2
3    <meta:directives namespace="http://namespaces.zope.org/smiley">
4
5      <meta:complexDirective
6          name="theme"
7          schema=".metadirectives.IThemeDirective"
8          handler=".metaconfigure.theme">
9
10       <meta:subdirective
11           name="smiley"
12           schema=".metadirectives.ISmileySubdirective" />
13
14     </meta:complexDirective>
15
16     <meta:directive
```

```
17          name="smiley"
18          schema=".metadirectives.ISmileyDirective"
19          handler=".metaconfigure.smiley" />
20
21      <meta:directive
22          name="defaultTheme"
23          schema=".metadirectives.IDefaultThemeDirective"
24          handler=".metaconfigure.defaultTheme" />
25
26    </meta:directives>
27
28  </configure>
```

Each meta directive, whether it is `directive`, `complexDirective` or `subdirective`, specifies the name of the directive and the schema it represents. The first two meta directives also take a `handler` attribute, which describes the callable object that will execute the directive.

You register this meta ZCML file with the system by placing a file called `smileyutility-meta.zcml` in the `package-includes` directory having the following content:

```
1 <include package="book.smileyutility" file="meta.zcml" />
```

29.6.4 (d) Test Directives

Now we are ready to test the directives. First we create a test ZCML file in `tests` called `smiley.zcml`. We write the directives in a way that we assume we are in the `tests` directory during its execution:

```
1  <configure
2      xmlns:zope="http://namespaces.zope.org/zope"
3      xmlns="http://namespaces.zope.org/smiley">
4
5    <zope:include package="book.smileyutility" file="meta.zcml" />
6
7    <theme name="yazoo">
8      <smiley text=":(" file="../smileys/yazoo/sad.png"/>
9      <smiley text=":)" file="../smileys/yazoo/smile.png"/>
10   </theme>
11
12   <theme name="plain" />
13
14   <smiley
15       theme="plain"
16       text=":("
17       file="../smileys/yazoo/sad.png"/>
18
19   <defaultTheme name="plain" />
20
21 </configure>
```

▷ Line 5: First read the meta configuration.

▷ Line 9–19: Use the three directives.

Now create a module called `test_directives.py` (the directive tests modules are usually called this way) and add the following test code:

```
1  import unittest
2
3  from zope.app import zapi
4  from zope.app.tests.placelesssetup import PlacelessSetup
5  from zope.configuration import xmlconfig
6
7  from book.smileyutility import tests
8  from book.smileyutility.interfaces import ISmileyTheme
9
10 class DirectivesTest(PlacelessSetup, unittest.TestCase):
11
12     def setUp(self):
13         super(DirectivesTest, self).setUp()
14         self.context = xmlconfig.file("smiley.zcml", tests)
15
16     def test_SmileyDirectives(self):
17         self.assertEqual(
18             zapi.getUtility(ISmileyTheme,
19                             'default')._GlobalSmileyTheme__smileys,
20             {u':(': u'/++resource++plain__sad.png'})
21         self.assertEqual(
22             zapi.getUtility(ISmileyTheme,
23                             'plain')._GlobalSmileyTheme__smileys,
24             {u':(': u'/++resource++plain__sad.png'})
25         self.assertEqual(
26             zapi.getUtility(ISmileyTheme,
27                             'yazoo')._GlobalSmileyTheme__smileys,
28             {u':)': u'/++resource++yazoo__smile.png',
29              u':(': u'/++resource++yazoo__sad.png'})
30
31     def test_defaultTheme(self):
32         self.assertEqual(zapi.getUtility(ISmileyTheme, 'default'),
33                          zapi.getUtility(ISmileyTheme, 'plain'))
34
35 def test_suite():
36     return unittest.TestSuite((
37         unittest.makeSuite(DirectivesTest),
38         ))
39
40 if __name__ == '__main__':
41     unittest.main()
```

As we can see, directive unittests can be very compact thanks to the `xmlconfig.file()` call.

▷ Line 4 & 10: Since we are registering resources during the configuration, we need to create a placeless setup.

▷ Line 14: Execute the configuration.

▷ Line 16–29: Make sure that all entries in the smiley themes were created.

▷ Line 31–33: A quick check that the default theme was set correctly.

▷ Line 35–41: This is just the necessary unittest boilerplate.

29.7 Step VI: Setting up some Smiley Themes

The service functionality is complete and we are now ready to hook it up to the system. We need to define the service and provide an implementation to the component architecture before we add two smiley themes. Therefore, in the `configure.zcml` file add:

```
1  <configure
2      xmlns="http://namespaces.zope.org/smiley"
3      i18n_domain="smileyutility">
4
5    <theme name="plain">
6      <smiley text=":("   file="./smileys/plain/sad.png"/>
7      <smiley text=":-("  file="./smileys/plain/sad.png"/>
8      <smiley text=":)"   file="./smileys/plain/smile.png"/>
9      <smiley text=":-)"  file="./smileys/plain/smile.png"/>
10     ...
11   </theme>
12
13   <theme name="yazoo">
14     <smiley text=":("   file="./smileys/yazoo/sad.png"/>
15     <smiley text=":-("  file="./smileys/yazoo/sad.png"/>
16     <smiley text=":)"   file="./smileys/yazoo/smile.png"/>
17     <smiley text=":-)"  file="./smileys/yazoo/smile.png"/>
18     ...
19   </theme>
20
21   <defaultTheme name="plain" />
22
23 </configure>
```

▷ Line 5–19: Provide two smiley themes. I abbreviated the list somewhat from the actual size, since I think you get the picture.

▷ Line 21: Set the default theme to "plain".

You can now activate the configuration by placing a file named `smileyutility-configure.zcml` in `package-includes`. It should have the following content:

```
1 <include package="book.smileyutility" />
```

29.8 Step VII: Integrate Smiley Themes into the Message Board

Okay, now we have these smiley themes, but we do not use them anywhere. So that it will be easier for us to see the smiley themes in action, I decided to extend the

messageboard example by yet another step. The new code consists of two parts: (a) allow the message board to select one of the available themes and (b) use smileys in the "Preview" tab of the message board.

29.8.1 (a) The Smiley Theme Selection Adapter

The additional functionality is best implemented using an Adapter and annotations. The interface that we need is trivially:

```
1 from zope.schema import Choice
2
3 class ISmileyThemeSpecification(Interface):
4
5     theme = Choice(
6         title=u"Smiley Theme",
7         description=u"The Smiley Theme used in message bodies.",
8         vocabulary=u"Smiley Themes",
9         default=u"default",
10        required=True)
```

Add this interface to the `interfaces.py` file of the message board. In the interface above we refer to a vocabulary called "Smiley Themes" without having specified it. We expect this vocabulary to provide a list of names of all available smiley themes. Luckily, creating vocabularies for utilities or utility names can be easily done using a single ZCML directive:

```
1 <vocabulary
2     name="Smiley Themes"
3     factory="zope.app.utility.vocabulary.UtilityVocabulary"
4     interface="book.smileyutility.interfaces.ISmileyTheme"
5     nameOnly="true" />
```

- ▷ Line 3: This is a special utility vocabulary class that is used to quickly create utility-based vocabularies.
- ▷ Line 4: This is the interface by which the utilities will be looked up.
- ▷ Line 5: If "nameOnly" is specified, the vocabulary will provide utility names instead of the utility component itself.

Next we create the adapter; place the following class into `messageboard.py`:

```
1 from zope.app.annotation.interfaces import IAnnotations
2 from book.messageboard.interfaces import ISmileyThemeSpecification
3
4 class SmileyThemeSpecification(object):
5
6     implements(ISmileyThemeSpecification)
7     __used_for__ = IMessageBoard
8
9     def __init__(self, context):
```

```
        self.context = self.__parent__ = context
        self._annotations = IAnnotations(context)
        if self._annotations.get(ThemeKey, None) is None:
            self._annotations[ThemeKey] = 'default'

    def getTheme(self):
        return self._annotations[ThemeKey]

    def setTheme(self, value):
        self._annotations[ThemeKey] = value

    # See .interfaces.ISmileyThemeSpecification
    theme = property(getTheme, setTheme)
```

As you can see, this is a very straightforward implementation of the interface using annotations and the adapter concept, both of which were introduced in the content components parts before.

The adapter registration and security is a bit tricky, since we must use a trusted adapter. It is not enough to just specify the "permission" attribute in the adapter directive, since it will only affect attribute access, but not mutation. Instead of specifying the "permission" attribute, we need to do a full security declaration using the `zope:class` and `zope:require` directives:

```
<class class=".messageboard.SmileyThemeSpecification">
  <require
      permission="book.messageboard.View"
      interface=".interfaces.ISmileyThemeSpecification"
      />
  <require
      permission="book.messageboard.Edit"
      set_schema=".interfaces.ISmileyThemeSpecification"
      />
</class>

<adapter
    factory=".messageboard.SmileyThemeSpecification"
    provides=".interfaces.ISmileyThemeSpecification"
    for=".interfaces.IMessageBoard"
    trusted="true" />
```

Last, we need to create a view to set the value. We can simply use the `browser:editform`. We configure the view with the following directive in `browser/configure.zcml`:

```
<editform
    name="smileyTheme.html"
    schema="book.messageboard.interfaces.ISmileyThemeSpecification"
    for="book.messageboard.interfaces.IMessageBoard"
    label="Change Smiley Theme"
    permission="book.messageboard.Edit"
    menu="zmi_views" title="Smiley Theme" />
```

By the way, the editform will automatically know how to look up the adapter and use it instead of the `MessageBoard` instance. If you now restart Zope 3, you should be able to change the theme to whatever you like.

29.8.2 (b) Using the Smiley Theme

The very final step is to use all this machinery. To do this, add a method called `body()` to the `MessageDetails` (`browser/message.py`) class:

```
1  def body(self):
2      """Return the body, but mark up smileys."""
3      body = self.context.body
4
5      # Find the messageboard and get the theme preference
6      obj = self.context
7      while not IMessageBoard.providedBy(obj) and \
8                obj is not None:
9          obj = zapi.getParent(obj)
10
11     if obj is None:
12         theme = None
13     else:
14         theme = ISmileyThemeSpecification(obj).theme
15
16     for text, url in getSmileysMapping(self.request, theme).items():
17         body = body.replace(
18             text,
19             '<img src="%s" label="%s"/>' %(url, text))
20
21     return body
```

▷ Line 5–14: This code finds the MessageBoard and, when found, gets the desired theme.

▷ Line 16–19: Using the theme, get the smiley mapping and convert one smiley after another from the text representation to an image tag referencing the smiley.

In the `details.pt` template, line 33, we now just have to change the call from `context/body` to `view/body` so that the above method is being used. Once you have done that you are ready to restart Zope 3 and enjoy the smileys.

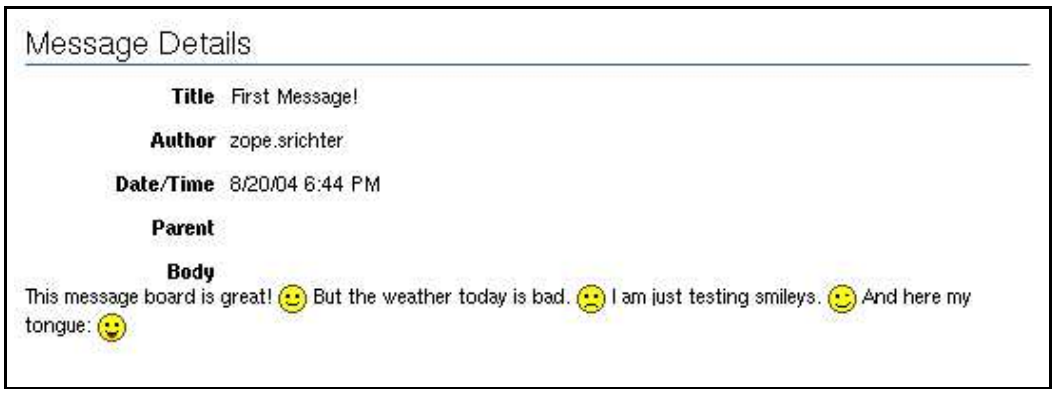


Figure 29.1: The message "Preview" screen featuring the smileys.

Exercises

1. The `GlobalSmileyTheme` currently is really only meant to work with resources that are defined while executing the `smiley` directives. However, it is not very hard to support general URLs as well. To do this, you will have to change the configuration to allow for another attribute called `url` and you have to adjust the theme so that it does not attempt all the time to add the root URL to the smiley path.

2. The directive tests do not check whether resources were created by the directives. You should enhance the `test_directives.py` module to do that.

3. I only updated one view to use the smileys. Update the skin as well to make use of them.

CHAPTER 30

LOCAL UTILITIES

Difficulty

Contributer

Skills

- Be comfortable with the Component Architecture, specifically utilities.
- Be familiar with the Site Manager Web GUI.
- Know the message board example as this affects somewhat the smiley support.
- You should be familiar with the global utility chapter, since this chapter creates its local companion.

Problem/Task

It is great to have our global smiley theme utilities. It works just fine. But what if I want to provide different icon themes for different message boards on the same Zope installation? Or I want to allow my online users to upload new themes? Then the global smiley theme is not sufficient anymore and we need a local and persistent version. This chapter will create a local smiley theme utility.

Solution

30.1 Introduction to Local Utilities

The semantics of local utilities are often very different than the ones for global utilities and have thus a very different implementation. For one, local utilities can be fully managed, which means that they can be added, edited and deleted. Only the first one of these actions is possible for their global counterparts. Furthermore, and most important, local utilities must know how to delegate requests to other utilities higher up, including the global version of the utility. All of these facts create a very different problem domain, which I intend to address in this chapter.

From the global smiley utility we already know that its purpose is to manage smileys (in a very limited way). Since the local smiley theme must be able to manage smileys fully, it is best to make the utility also a container that can only contain smileys. A smiley component will simply be a glorified image that can only be contained by local smiley themes. Thus we need to develop an `ILocalSmileyTheme` that extends `IContainer`. This interface must also limit its containable items to be only smileys. The second interface will be `ISmiley`, which simply extends `IImage` and only allows itself to be added to smiley themes.

Like all other local components, local utilities must be registered. One way would be to write a custom registration component; however, the simpler method is to have the local smiley theme provide the `ILocalUtility` marker interface. A registration component (including all necessary views) exists for any object that provides this interface making the implementation of a local utility much simpler.

30.2 Step I: Defining Interfaces

As I pointed out in the introduction, we will need two new interfaces. The first one is the `ISmiley` interface:

```
1 from zope.schema import Field
2
3 from zope.app.container.constraints import ContainerTypesConstraint
4 from zope.app.file.interfaces import IImage
5
6 class ISmiley(IImage):
7     """A smiley is just a glorified image"""
8     __parent__ = Field(
9         constraint = ContainerTypesConstraint(ISmileyTheme))
```

As I said before, the smiley component is simply an image that can only be added to smiley themes. The second interface is the `ILocalSmileyTheme`, which will manage all its smileys in a typical container-like fashion:

```
1 from zope.app.container.constraints import ItemTypePrecondition
2 from zope.app.container.interfaces import IContainer
3
```

```
class ILocalSmileyTheme(ISmileyTheme, IContainer):
    """A local smiley themes that manages its smileys via the container API"""

    def __setitem__(name, object):
        """Add a IMessage object."""

    __setitem__.precondition = ItemTypePrecondition(ISmiley)
```

After we make the local smiley theme a container, we declare that it can only contain smileys. If you do not know about preconditions and constraints in interfaces, please read the chapter on creating a content component.

30.3 Step II: Implementation

Implementing the smiley is trivial. In a new Python file called `localtheme.py` add the following:

```
from zope.app.file.image import Image
from interfaces import ISmiley

class Smiley(Image):
    implements(ISmiley)
```

Now we just need to provide an implementation for the theme. As before, we can use the `BTreeContainer` as a base class that provides us with a full implementation of the `IContainer` interface. Then all that we have to worry about are the three `ISmileyTheme` API methods.

```
from zope.component.exceptions import ComponentLookupError
from zope.interface import implements

from zope.app import zapi
from zope.app.container.btree import BTreeContainer
from zope.app.component.localservice import getNextService

from interfaces import ISmileyTheme, ILocalSmileyTheme

class SmileyTheme(BTreeContainer):
    """A local smiley theme implementation."""
    implements(ILocalSmileyTheme)

    def getSmiley(self, text, request):
        "See book.smileyutility.interfaces.ISmileyTheme"
        smiley = self.querySmiley(text, request)
        if smiley is None:
            raise ComponentLookupError, 'Smiley not found.'
        return smiley

    def querySmiley(self, text, request, default=None):
        "See book.smileyutility.interfaces.ISmileyTheme"
        if text not in self:
            theme = queryNextTheme(self, zapi.name(self))
            if theme is None:
                return default
```

```
27              else:
28                  return theme.querySmiley(text, request, default)
29          return getURL(self[text], request)
30
31      def getSmileysMapping(self, request):
32          "See book.smileyutility.interfaces.ISmileyTheme"
33          theme = queryNextTheme(self, zapi.name(self))
34          if theme is None:
35              smileys = {}
36          else:
37              smileys = theme.getSmileysMapping(request)
38
39          for name, smiley in self.items():
40              smileys[name] = getURL(smiley, request)
41
42          return smileys
43
44
45  def queryNextTheme(context, name, default=None):
46      """Get the next theme higher up."""
47      theme = default
48      while theme is default:
49          utilities = queryNextService(context, zapi.servicenames.Utilities)
50          if utilities is None:
51              return default
52          theme = utilities.queryUtility(ISmileyTheme, name, default)
53          context = utilities
54      return theme
55
56  def getURL(smiley, request):
57      """Get the URL of the smiley."""
58      url = zapi.getView(smiley, 'absolute_url', request=request)
59      return url()
```

▷ Line 14–19: This implementation is identical to the global one. We have the method `querySmiley()` do the work.

▷ Line 21–29: If the requested smiley is available in the theme, simply return its URL. However, if the smiley is not found, we should not give up that quickly. It might be defined in a theme (with the same name) in a level higher up. The highest layer are the global components. If a theme of the same name exists higher up, then try to get the smiley from there. If no such theme exists, then its time to give up and to return the default value.

This generalizes very nicely to all local components. Local components should only concerned with querying and searching their local place and not stretch out into other places. For utilities, the request should then always be forwarded to the next occurrence at a higher place. This method will automatically be able to recursively search the entire path all the way up. The termination condition is usually the global utility, which always has to return and will never refer to another place. If you do not have a global version of the utility available, then you

need to put a condition in your local code, terminating when no other utility is found.

- Line 31–42: This is method that has to be very careful about the procedure it uses to generate the result. The exact same smiley (text, theme) might be declared in several locations along the path, but only the last declaration (closest to the current location) should make it into the smiley mapping. Therefore we *first* get the *acquired* results and then merge the local smiley mapping into it, so that the local smileys are always added last. Note that this implementation makes this method also recursive, ensuring that all themes with the matching name are considered.

- Line 34–43: This method returns the next matching theme up. Starting at `context`, the `queryNextService()` method walks up the tree looking for the next site in the path. If a site is found, it sees whether it finds the specified service (in our case the utility service) in the site. If not, it keeps walking. It will terminate its search once the global site is reached (`None` is returned) or a service is found.

  If the utilities service was found, we now need to ensure that it also has a matching theme. If not we have to keep looking by finding the next utilities service. If a matching theme is found, the `while` loop's condition is fulfilled and the theme is returned.

- Line 45–48: Since smiley entries are not URLs in the local theme, we look up their URLs using the `absolute_url` view.

As you can see, the implementation of the local theme was a bit more involved, since we had to worry about the delegation of the requests. But it is downhill from now on. What we got for free was a full management and registration user and programming interface for the local themes and smileys, which is the equivalent of the ZCML directives we had to develop for the global theme.

Until now we always wrote the tests right after the implementation. However, tests for local components very much reflect their behavior in the system and the tests will be easier to understand, if we get the everything working first. Therefore, we will next develop the necessary registrations followed by providing some views.

30.4 Step III: Registrations

First we register the local theme as a new content type and local utility. Making it a local utility will also ensure that it can only be added to site management folders. Add the following directives to you configuration file:

```
1  <zope:content class=".localtheme.SmileyTheme">
2    <zope:factory
3        id="book.smileyutility.SmileyTheme"
4        title="Smiley Theme"
5        description="A Smiley Theme"
6        />
7    <zope:implements
8        interface="zope.app.utility.interfaces.ILocalUtility"
9        />
10   <zope:implements
11       interface="zope.app.container.interfaces.IContentContainer"
12       />
13   <zope:implements
14       interface="zope.app.annotation.interfaces.IAttributeAnnotatable"
15       />
16   <zope:allow
17       interface="zope.app.container.interfaces.IReadContainer"
18       />
19   <zope:require
20       permission="zope.ManageServices"
21       interface="zope.app.container.interfaces.IWriteContainer"
22       />
23   <zope:allow
24       interface=".interfaces.ISmileyTheme"
25       />
26 </zope:content>
```

General Note: The reason we use the `zope:` prefix in our directives here is that we used the `smiley` namespace as the default.

▷ Line 7–9: Declare the local theme component to be a local utility. Since this is just a marker interface, no special methods or attributes must be implemented.

▷ Line 10–12: In order for the precondition of `__setitem__()` to work, we need to make the smiley theme also a `IContentContainer`. This is just another marker interface.

▷ Line 13–15: All local components should be annotatable, so that we can append Dublin Core and other meta-data.

▷ Line 16–18: Allow everyone to just access the smileys at their heart's content.

▷ Line 19–22: However, for changing the theme we require the service management permission.

▷ Line 24–27: We also want to make the theme's API methods publicly available.

Now that we just have to declare the `Smiley` class as a content type.

```
1 <zope:content class=".localtheme.Smiley">
2   <zope:require
3       like_class="zope.app.file.image.Image"
4       />
5 </zope:content>
```

▷ Line 2–4: Just give the Smiley the same security declarations as the image. Since the smiley does not declare any new methods and attributes, we have to make no further security declarations.

The components are registered now, but we will still not be able to do much, since we have not added any menu items to the add menu or any other management view.

30.5 Step IV: Views

As you will see, the browser code for the theme is minimal, so that we will not create a separate `browser` package and we place the browser code simply in the main configuration file. As always, you need to add the `browser` namespace first:

```
1 xmlns:browser="http://namespaces.zope.org/browser"
```

Now we create add menu entries for each content type.

```
1  <browser:addMenuItem
2      class=".localtheme.Smiley"
3      title="Smiley"
4      description="A Smiley"
5      permission="zope.ManageServices"
6      />
7
8  <browser:addMenuItem
9      class=".localtheme.SmileyTheme"
10     title="Smiley Theme"
11     description="A Smiley Theme"
12     permission="zope.ManageServices"
13     />
```

We also want the standard container management screens be available in the theme, so we just add the following directive:

```
1  <browser:containerViews
2      for=".localtheme.SmileyTheme"
3      index="zope.View"
4      contents="zope.ManageServices"
5      add="zope.ManageServices"
6      />
```

Practically, you can now restart Zope 3 and test the utility and everything should work as expected. Even so, I want to create a couple more convenience views that make the utility a little bit nicer.

First, you might have noticed already the "Tools" tab in the site manager. Tools are mainly meant to make the management of utilities simpler; and the best about it is that a tools entry requires only one simple directive:

```
1  <browser:tool
2      interface=".interfaces.ISmileyTheme"
3      title="Smiley Themes"
```

```
4     description="Smiley Themes allow you to convert text-based to icon-based
5     smileys."
6     />
```

▷ Line 1: Since tools are not components, but just views on the site manager, the directive is part of the `browser` namespace.

▷ Line 2: This is the interface under which the utility is registered.

▷ Line 3–4: Here we provide a human-readable title and description for the tool, which is used in the tools overview screen.

The second step is to create a nice "Overview" screen that tells us the available local and acquired smileys available for a particular theme. The first step is to create a view class, which provides one method for retrieving all locally defined smileys and one method that retrieves all acquired smileys from higher up themes. In a new file called `browser.py` add the following code:

```
1  from zope.app import zapi
2
3  from localtheme import queryNextTheme, getURL
4
5  class Overview(object):
6
7      def getLocalSmileys(self):
8          return [{'text': name, 'url': getURL(smiley, self.request)}
9                  for name, smiley in self.context.items()]
10
11     def getAcquiredSmileys(self):
12         theme = queryNextTheme(self.context, zapi.name(self.context))
13         map = theme.getSmileysMapping(self.request)
14         return [{'text': name, 'url': path} for name, path in map.items()
15                 if name not in self.context]
```

▷ Line 7–9: Getting all the locally defined smileys is easy; simply get all the items from the container and convert the smiley object to a URL. The return object will be a list of dictionaries of the following form:

- "text" ⟶ This is the text representation of the smiley; in this case the name of the smiley object.
- "url" ⟶ This is the URL of the smiley as located in the theme. We already developed a function for getting the URL (`getURL()`), so let's reuse it.

▷ Line 11–15: We know that `getSmileysMapping()` will get us all local and acquired smileys. But if we get the next theme first and then call the method, we will only get the acquired smileys with respect to this theme. We only need to make sure that we exclude smileys that are also defined locally. From the mapping, we then create the same output dictionary as in the previous function.

The template that will make use of the two view methods above could look something like this:

```
<html metal:use-macro="views/standard_macros/view">
<head>
  <title metal:fill-slot="title"
         i18n:translate="">Smiley Theme</title>
</head>
<body>
<div metal:fill-slot="body">

  <h2 i18n:translate="">Local Smileys</h2>
  <ul>
    <li tal:repeat="smiley view/getLocalSmileys">
    <b tal:content="smiley/text"/> &#8594;
    <img src="" tal:attributes="src smiley/url"/>
    </li>
  </ul>

  <h2 i18n:translate="">Acquired Smileys</h2>
  <ul>
    <li tal:repeat="smiley view/getAcquiredSmileys">
    <b tal:content="smiley/text"/> &#8594;
    <img src="" tal:attributes="src smiley/url"/>
    </li>
  </ul>

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

Place the above template in a new file called `overview.pt`. All that's left now is to register the view using a simple `browser:page` directive.

```
<browser:page
    name="overview.html"
    menu="zmi_views" title="Overview"
    for=".localtheme.SmileyTheme"
    permission="zope.ManageServices"
    class=".browser.Overview"
    template="overview.pt" />
```

30.6 Step V: Working with the Local Smiley Theme

Let's test the new local theme now by walking through the steps of creating a utility via the Web interface. This will help us understand the tests we will have to write at the end. First restart Zope 3 and log in with a user that is also a manager. Go to the contents view of the root folder and click on the "Manage Site" link just below the tabs. When the screen is loaded, click on the "Tools" tab and choose the "Smiley Themes" tool. You can now add a new theme by pressing the "Add" button. Once the new page appears, enter the name of the theme in the text field and press the "Add" button. You best choose a name for the theme that is already used as a

global theme as well, like "plain". This way we can test the acquisition of themes better. Once the browser is done loading the following page, you should be back in the smiley themes tool overview screen listing the "plain" theme, which is already registered as being "active".

Figure 30.1: An overview of all smiley themes.

To add a new smiley click on "plain", which will bring you to the theme's "Contents" view. Right beside the "Add" button you will see a text field. Enter the name ":-)" there and press "Add". You now created a new smiley. Click on ":-)" to upload a new image. Choose an image in the "Data" row and press "Change", which will upload the image. Repeat the procedure for the ":)" smiley. To see the contrast, you might want to upload smileys from the "yazoo" theme.

Once you are done, click on the "Overview" tab and you should see the two local and a bunch of acquired smileys, which are provided by the global "plain" smiley theme.

If you like you can now go to the message board and ensure that the local smiley definitions are now preferred over the global ones for the "plain" theme.

30.7 Step VI: Writing Tests

While we have a working system now, we still should write tests, so that we can figure out whether all aspects of the local smiley theme are working correctly. The truly interesting part about testing any local component is the setup; once you get this right, the tests are quickly written.

When testing local components one must basically bring up an entire bootstrap ZODB with folders and site managers. Luckily, there are some very nice utility functions that help with this tedious setup. They can be found in `zope.app.tests.setup`. Here are the functions that are commonly useful to the developer:

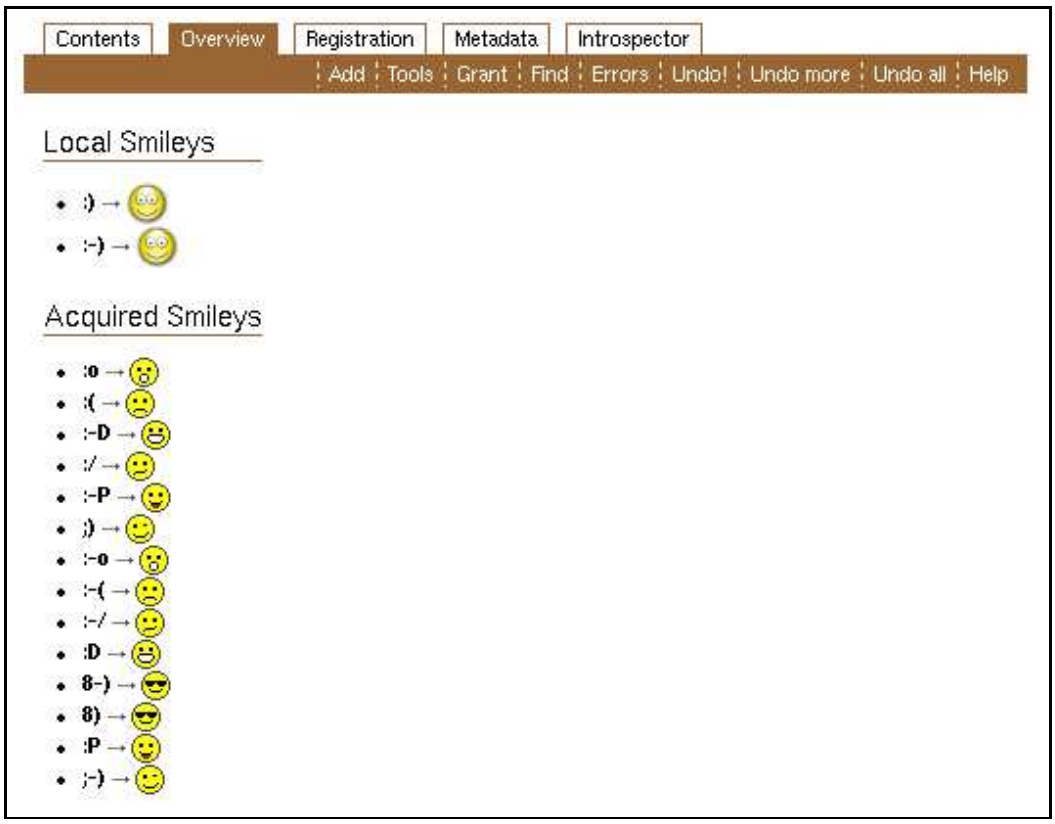


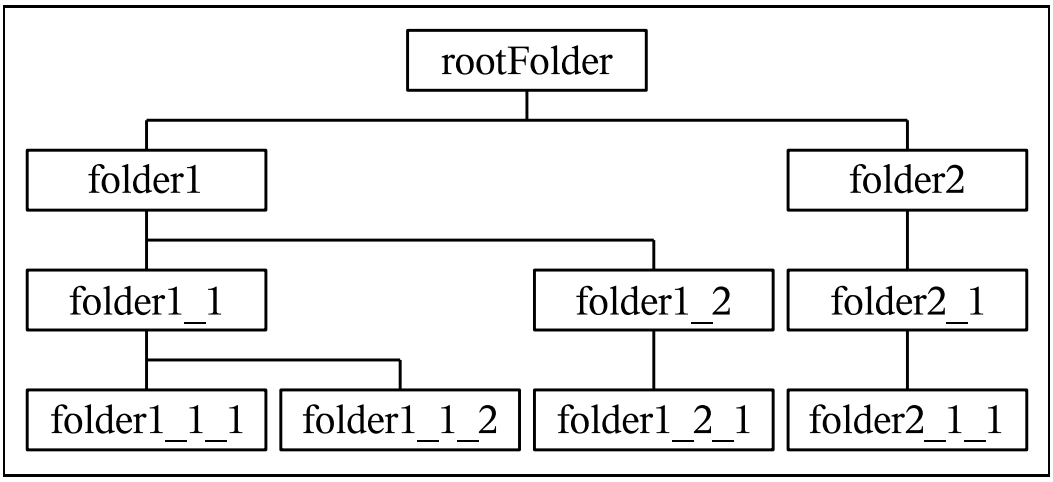
Figure 30.2: An overview of all available smileys in this theme.

- `setUpAnnotations()`: This function registers the attribute annotations adapter. This function is also useful for placeless setups.

- `setUpTraversal()`: This function sets up a wide range of traversal-related adapters and views, including everything that is needed to traverse a path, get object's parent path and traverse the "etc" namespace. The `absolute_url` view is also registered.

- `placefulSetUp(site=False)`: Like the placeless setup, this function registers all the interfaces and adapters required for doing anything useful. Included are annotations, dependency framework, traversal hooks and the registration machinery. If `site` is set to True, then a root folder with a `ServiceManager` (also known as site manager) inside will be created and the site manager is returned.

- `placefulTearDown()`: Like the placeless equivalent, this function correctly shuts down the registries.
- `buildSampleFolderTree()`: A sample folder tree is built to support multi-place settings, something that is important for testing acquisition of local components. The following structure is created:



- `createServiceManager(folder,setsite=False)`: Create a local service/site manager for this folder. Note that the function can be used for any object that implements `ISite`. If `setsite` is True, then the thread global site variable will be set to the new site as well.
- `addService(servicemanager,name,service,suffix="")`: This function adds a service instance to the specified service manager and registers it. The service will be available as `name` and the instance will be stored as `name+suffix`.
- `addService(servicemanager,name,iface,utility,suffix="")`: Her we register a utility providing the interface `iface` and having the name `name` to the specified service manager. The utility will be stored in the site management folder under the name `name+suffix`.

Now we are ready to look into writing our tests. Like for the global theme, I decided to write doc tests for the theme. Thus, add the following lines in `tests/test_doc.py` after line 41:

```
1 DocTestSuite('book.smileyutility.localtheme')
```

The following tests will all be added to the doc string or the `SmileyTheme` class. We begin with calling the `placefulSetUp()` function and setting up the folder tree.

```
1 >>> from zope.app.tests import setup
2 >>> from zope.app.utility.utility import LocalUtilityService
```

```
>>> site = setup.placefulSetUp()
>>> rootFolder = setup.buildSampleFolderTree()
```

Next we write a convenience function that let's us quickly add a new smiley to a local theme.

```
Setup a simple function to add local smileys to a theme.

>>> import os
>>> import book.smileyutility
>>> def addSmiley(theme, text, filename):
...     base_dir = os.path.dirname(book.smileyutility.__file__)
...     filename = os.path.join(base_dir, filename)
...     theme[text] = Smiley(open(filename, 'r'))
```

Now that the framework is all setup, we can add some smiley themes in various folders.

```
Create components in root folder

>>> site = setup.createServiceManager(rootFolder)
>>> utils = setup.addService(site, zapi.servicenames.Utilities,
...                          LocalUtilityService())
>>> theme = setup.addUtility(site, 'plain', ISmileyTheme, SmileyTheme())
>>> addSmiley(theme, ':)',  'smileys/plain/smile.png')
>>> addSmiley(theme, ':(',  'smileys/plain/sad.png')

Create components in `folder1`

>>> site = setup.createServiceManager(rootFolder['folder1'])
>>> utils = setup.addService(site, zapi.servicenames.Utilities,
...                          LocalUtilityService())
>>> theme = setup.addUtility(site, 'plain', ISmileyTheme, SmileyTheme())
>>> addSmiley(theme, ':)',  'smileys/plain/biggrin.png')
>>> addSmiley(theme, '8)',  'smileys/plain/cool.png')
```

- ▷ Line 3: First, we make the root folder a site.
- ▷ Line 4–5: There are no local services in a new site by default. Before we can add utilities, we first need to add a local utility service to the site.
- ▷ Line 6–8: First we create the theme and add it as a utility to the site. Then just add two smileys to it.
- ▷ Line 10–17: A setup similar to the root folder for `folder1`.

Now we have completely setup the system and can test the API methods. First, let's test the `getSmiley()` and `querySmiley()` methods via the package's API convenience functions.

```
Now test the single smiley accessor methods

>>> from zope.publisher.browser import TestRequest
>>> from zope.app.component.localservice import setSite
```

```
5  >>> from book.smileyutility import getSmiley, querySmiley
6
7  >>> setSite(rootFolder)
8  >>> getSmiley(':)', TestRequest(), 'plain')
9  'http://127.0.0.1/++etc++site/default/plain/%3A%29'
10 >>> getSmiley(':(', TestRequest(), 'plain')
11 'http://127.0.0.1/++etc++site/default/plain/%3A%28'
12 >>> getSmiley('8)', TestRequest(), 'plain')
13 Traceback (most recent call last):
14 ...
15 ComponentLookupError: 'Smiley not found.'
16 >>> querySmiley('8)', TestRequest(), 'plain', 'nothing')
17 'nothing'
18
19 >>> setSite(rootFolder['folder1'])
20 >>> getSmiley(':)', TestRequest(), 'plain')
21 'http://127.0.0.1/folder1/++etc++site/default/plain/%3A%29'
22 >>> getSmiley(':(', TestRequest(), 'plain')
23 'http://127.0.0.1/++etc++site/default/plain/%3A%28'
24 >>> getSmiley('8)', TestRequest(), 'plain')
25 'http://127.0.0.1/folder1/++etc++site/default/plain/8%29'
26 >>> getSmiley(':|', TestRequest(), 'plain')
27 Traceback (most recent call last):
28 ...
29 ComponentLookupError: 'Smiley not found.'
30 >>> querySmiley(':|', TestRequest(), 'plain', 'nothing')
31 'nothing'
```

- ▷ Line 7: Set the current site to the root folder. All requests are now with respect from that site.
- ▷ Line 8–11: Make sure that the basic local access works. Note that the `TestRequest` defines the computers IP address to be `127.0.0.1` and is not computer-specific.
- ▷ Line 12–17: Make sure that a `ComponentLookupError` is raised, if a smiley is not found or the `default` is returned, if `querySmiley()` was used.
- ▷ Line 19–31: Repeat the tests for using `folder1` as location. Specifically interesting is line 22–23, since the smiley is not found locally, but retrieved from the root folder's theme.

Let's now test the 'getSmileysMapping()' method. To do that we create a small helper method that helps us compare dictionaries.

```
1  >>> from pprint import pprint
2  >>> from book.smileyutility import getSmileysMapping
3  >>> def output(dict):
4  ...     items = dict.items()
5  ...     items.sort()
6  ...     pprint(items)
7
8  >>> setSite(rootFolder)
```

```
>>> output(getSmileysMapping(TestRequest(), 'plain'))
[(u':(', 'http://127.0.0.1/++etc++site/default/plain/%3A%28'),
 (u':)', 'http://127.0.0.1/++etc++site/default/plain/%3A%29')]

>>> setSite(rootFolder['folder1'])
>>> output(getSmileysMapping(TestRequest(), 'plain'))
[(u'8)', 'http://127.0.0.1/folder1/++etc++site/default/plain/8%29'),
 (u':(', 'http://127.0.0.1/++etc++site/default/plain/%3A%28'),
 (u':)', 'http://127.0.0.1/folder1/++etc++site/default/plain/%3A%29')]
>>> getSmileysMapping(TestRequest(), 'foobar')
Traceback (most recent call last):
...
ComponentLookupError: \
(<InterfaceClass book.smileyutility.interfaces.ISmileyTheme>, 'foobar')
```

▷ Line 8–17: Again, test the method for two locations, so that acquisition can be tested.

▷ Line 18–22: Make sure we do not accidently find any non-existent themes.

After all the tests are complete, we need to cleanly shutdown the test case.

```
>>> setup.placefulTearDown()
```

You should now run the tests and see that they all pass. Another interesting function that deserves careful testing is the `queryNextTheme()`. I will not explain the test here, since it is very similar to the previous one and will ask you to look in the code yourself for the test or even try to develop it yourself.

Exercises

1. Something that I have silently ignored is to allow to specify a default smiley theme. This can be simply accomplished by adding a second registration for a theme. Implement this feature.

2. Currently, smileys are always acquired. But this might be sometimes undesired and should be really up to the manager to decide. Develop an option that allows the manager to choose whether smileys should be acquired or not.

3. Uploading one smiley at a time might be extremely tedious. Instead, it should be allowed to upload ZIP-archives that contain the smileys. Implement that feature.

CHAPTER 31

VOCABULARIES AND RELATED FIELDS/WIDGETS

Difficulty

Sprinter

Skills

- Be familiar with the `zope.schema` package.
- Be familiar with the `zope.app.widget` package.

Problem/Task

You will agree that schemas in combination with widgets and forms are pretty cool. The times of writing boring HTML forms and data verification are over. However, the standard fields make it hard (if not impossible) to create a dynamic list of possible values to choose from. To solve this problem, the vocabulary and their corresponding fields and widgets were developed. In this chapter we will demonstrate one of the common usages of vocabularies in Zope 3.

Solution

31.1 Introduction

A common user interface pattern is to provide the user with a list of available or possible values from which one or more might be selected. This is done to reduce

the amount of errors the user could possibly make. Often the list of choices is static, meaning they do not change over time or are dependent on a particular situation. On the other hand, you commonly have choices for a field that depends strongly on the situation you are presented with.

The `Choice` field is used to allow users to select a value from a provided list. If you pass the keyword argument `values` to the list, the field let's you always choose from this static list of values. However, if you specify a vocabulary via the `vocabulary` argument, then it will be used to provide the list of available choices. By the way, the argument either accepts a vocabulary object or a vocabulary name (string). If you wish to select multiple items from a list of choices, then you can either use the `Tuple`, `List` or `Set` field, which except a `value_type` argument, which specifies the type of values that can reside in these collection types. If you pass a `Choice` field as `value_type`, then a widget will be chosen that let's you select only from the choices in the `Choice` field.

Vocabularies in themselves are not difficult to understand, but their application ranges from the generation of a static list of elements to providing a list of all available RDB connections, for example. But at the end of the day, vocabularies just provide a list of iterms or terms, which is the correct jargon. For large data sets vocabularies also have a simple query support, so that we can build a sane user interface for the data; however, the default widgets do not support queries that well yet.

Generally there are two scenarios of vocabulary usage in Zope 3: the ones that so and others that do need a place to generate the list of terms. Vocabularies that do not need a place can be created as singeltons and would be useful when data is retrieved from a file, RDB or any other Zope-external data source. In this chapter, however, we are going to implement a vocabulary that provides a list of all items in a container (or any other `IReadMapping` object). Thus the location clearly matters.

Vocabularies that need a location, cannot exist as singletons, but the location must be passed into the constructor. Zope 3 provides a vocabulary registry with which one can register vocabulary factories (which are usually just the classes) by name. The ZCML directive, `zope:vocabulary`, can be used as follows:

```
1 <vocabulary
2    name="VocabularyName"
3    factory=".vocab.Vocabulary" />
```

You can then use the vocabulary in a schema by declaring a `Choice` field:

```
1 from zope.schema import Choice
2
3 field = Choice(
4     title=u"...",
5     description=u"...",
6     vocabulary="VocabularyName")
```

If the `vocabulary` argument value is a string, then it is used as a vocabulary name, and the vocabulary is created with a context whenever needed. But the argument also accepts `IVocabulary` instances, which are directly used.

Okay, I wrote already too much. Let's see how we can achieve our task using vocabularies.

31.2 Step I: The Vocabulary and its Terms

A vocabulary has a very simple interface. It is almost like a simple mapping object with some additional functionality. The main idea is that a vocabulary provides `ITerm` objects. A term has simply a `value` that can be any Python object. However, for Web forms (and other user interfaces) this minimalistic interface does not suffice, since we have no way of reliably specifiying a unique id (a string) for a term, which we need to do to create any HTML input element with the terms. To solve this problem, the `ITokenizedTerm` was developed, which provides a `token` attribute that must be a string uniquely identifying the term.

Since our vocabulary deals with folder item names, our `ITerm` `value` is equal to the `token`. Therefore, we only need a minimal implementation of `ITokenizedTerm` as seen below.

```
from zope.interface import implements
from zope.schema.interfaces import ITokenizedTerm

class ItemTerm(object):
    """A simple term implementation for items."""
    implements(ITokenizedTerm)
    def __init__(self, value):
        self.value = self.token = value
```

Create a new package called `itemvocabulary` in `ZOPE3/src/book` and place the above code in the `__init__.py` file. Next we need to implement the vocabulary. Since the context of the vocabulary is an `IReadMapping` object, the implementation is straightforward:

```
from zope.schema.interfaces import IVocabulary, IVocabularyTokenized
from zope.interface.common.mapping import IEnumerableMapping

class ItemVocabulary(object):
    """A vocabulary that provides the keys of any IEnumerableMapping object.

    Every dictionary will qualify for this vocabulary."""
    implements(IVocabulary, IVocabularyTokenized)
    __used_for__ = IEnumerableMapping

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

    def __iter__(self):
```

```
15          """See zope.schema.interfaces.IIterableVocabulary"""
16          return iter([ItemTerm(key) for key in self.context.keys()])
17
18      def __len__(self):
19          """See zope.schema.interfaces.IIterableVocabulary"""
20          return len(self.context)
21
22      def __contains__(self, value):
23          """See zope.schema.interfaces.IBaseVocabulary"""
24          return value in self.context
25
26      def getQuery(self):
27          """See zope.schema.interfaces.IBaseVocabulary"""
28          return None
29
30      def getTerm(self, value):
31          """See zope.schema.interfaces.IBaseVocabulary"""
32          if value not in self.context:
33              raise LookupError, value
34          return ItemTerm(value)
35
36      def getTermByToken(self, token):
37          """See zope.schema.interfaces.IVocabularyTokenized"""
38          return self.getTerm(token)
```

▷ Line 8: Make sure that you implement both, `IVocabulary` and `IVocabularyTokenized`, so that the widget mechanism will work correctly later.

▷ Line 14–16: Make sure that the values of the iterator are `ITokenizedTerm` objects and not simple strings. If you only implement `IVocabulary`, then the objects just have to implement `ITerm`.

▷ Line 26–28: We do not support queries in this implementation. The interface specifies that vocabularies not supporting queries must return `None`.

▷ Line 30–34: We must be careful here and not just create an `ItemTerm` from the value, since the interface specifies that if the value is not available in the vocabulary, a `LookupError` should be raised.

▷ Line 36–38: Since the `token` and the `value` are equal, we can just forward the request to `getTerm()`.

Since the vocabulary requires a context for initiation, we need to register it with the vocabulary registry. The vocabulary is also used in untrusted environments, so that we have to make security assertions for it and the term. Place the ZCML directives below in the `configure.zcml` of the package.

```
1 <configure
2     xmlns="http://namespaces.zope.org/zope"
3     i18n_domain="itemvocabulary">
4
```

```
5  <vocabulary
6      name="Items"
7      factory=".ItemVocabulary" />
8
9  <content class=".ItemVocabulary">
10   <allow interface="zope.schema.interfaces.IVocabulary"/>
11   <allow interface="zope.schema.interfaces.IVocabularyTokenized"/>
12 </content>
13
14 <content class=".ItemTerm">
15   <allow interface="zope.schema.interfaces.ITokenizedTerm"/>
16 </content>
17
18 </configure>
```

▷ Line 5–7: Register the vocabulary under the name "Items". The vocabulary directive is available in the default "zope" namespace.

▷ Line 9–16: We simply open up all of the interfaces to the public, since the objects that provide the data are protected themselves.

That was easy, right? Now, let's write some quick tests for this code.

31.3 Step II: Testing the Vocabulary

The tests are as straightforward as the code itself. We are going to only test the vocabulary, since it uses the trivial term. In the doc string of the `ItemVocabulary` class add the following example and test code:

```
1  Example:
2
3  >>> data = {'a': 'Anton', 'b': 'Berta', 'c': 'Charlie'}
4  >>> vocab = ItemVocabulary(data)
5  >>> iterator = iter(vocab)
6  >>> iterator.next().token
7  'a'
8  >>> len(vocab)
9  3
10 >>> 'c' in vocab
11 True
12 >>> vocab.getQuery() is None
13 True
14 >>> vocab.getTerm('b').value
15 'b'
16 >>> vocab.getTerm('d')
17 Traceback (most recent call last):
18 ...
19 LookupError: d
20 >>> vocab.getTermByToken('b').token
21 'b'
22 >>> vocab.getTermByToken('d')
23 Traceback (most recent call last):
```

```
...
LookupError: d
```

Note that we can simply use a dictionary as our test context, since it fully provides `IEnumerableMapping`. The tests are activated via a doc test that is initialized in `tests.py` with the following code:

```
import unittest
from zope.testing.doctestunit import DocTestSuite

def test_suite():
    return unittest.TestSuite((
        DocTestSuite(book.itemvocabulary'),
        ))

if __name__ == '__main__':
    unittest.main(defaultTest='test_suite')
```

You can execute the tests as usual via the Zope 3 test runner or call the test file directly after you have set the correct Python path.

31.4 Step III: The Default Item Folder

To see the vocabulary working, we will develop a special folder that simply keeps track of a default item (whatever "default" may mean). Since the folder is part of a browser demonstration, we place the folder interface and implementation in the file `browser.py`:

```
from zope.interface import implements, Interface
from zope.schema import Choice
from zope.app.folder import Folder

class IDefaultItem(Interface):

    default = Choice(
        title=u"Default Item Key",
        description=u"Key of the default item in the folder.",
        vocabulary="Items")

class DefaultItemFolder(Folder):
    implements(IDefaultItem)

    default = None
```

- ▷ Line 7–10: Here you can see the `Choice` field in a very common setup and usage. The `vocabulary` argument can either be the vocabulary name or a vocabulary instance, as pointed out earlier in this chapter.
- ▷ Line 12–15: A trivial content component implementation that combines `IFolder` and `IDefaultItem`.

Now we only have we just have to register the new content component, make some security assertions and create an edit form for the `default` value. All of this can be done with the following three ZCML directives:

```
<content class=".browser.DefaultItemFolder">
  <require like_class="zope.app.folder.Folder"/>

  <require
      permission="zope.View"
      interface=".browser.IDefaultItem" />

  <require
      permission="zope.ManageContent"
      set_schema=".browser.IDefaultItem" />
</content>

<browser:addMenuItem
    class=".browser.DefaultItemFolder"
    title="Default Item Folder"
    permission="zope.ManageContent" />

<browser:editform
    schema=".browser.IDefaultItem"
    for=".browser.IDefaultItem"
    label="Change Default Item"
    name="defaultItem.html"
    permission="zope.ManageContent"
    menu="zmi_views" title="Default Item" />
```

Don't forget to register the `browser` namespace in the `configure` tag:

```
xmlns:browser="http://namespaces.zope.org/browser"
```

Finally, you have to tell the system about the new paclage, so that it will read its configuration. Place a file called `itemvocabulary-configure.zcml` in the `package-includes` directory having the following one line directive:

```
<include package="book.itemvocabulary" />
```

You are now ready to go. Restart Zope 3. Once you refresh the ZMI, you will see that you can now add a "Default Item Folder". Create such a folder and add a couple other components to it, like images and files. If you now click on the "Default Item" tab, you will see a selection box with the names of all contained objects. Select one and submit the form. You now stored the name of the object that will be considered the "default". As you can see, there exist widgets that know how to display a vocabulary field. See exercise 1 for changing the used widget.

Exercises

1. Change the `defaultItem.html` of the `DefaultItemFolder` so that it uses radio buttons instead of a drop-down menu.

CHAPTER 32

EXCEPTION VIEWS

Difficulty

Newcomer

Skills

- You should be knowledgeable about writing page templates.
- Have some basic ZCML knowledge.

Problem/Task

Zope 3 has the capability to provide views for exceptions and errors. Zope already provides views for some of the most common user errors, such as `NotFound` (a page was not found), and even a generic view for all exceptions. However, when you have a specific application error, you usually want to provide a customized error message.

Solution

32.1 Introduction

Exceptions are a powerful tool in programming. However, sometimes it becomes hard to deal with them when it comes to the point that exceptions reach the user. In Zope 3 we allow exceptions to have views, so that the user will always see a very friendly message when an error occurred. Thereby we clearly differentiate between

errors that were raised due to a programming error (a bug) and errors that were raised on purpose to signalize a user error.

Programming errors should never occur in a production-quality application, and as Jim Fulton said: "I want to discourage people from trying to make *all errors* look good" (my emphasis). Thus Zope 3 provides by default only a very minimalistic view saying "System Error". An exception to that is the view for the `NotFoundError`, which displays a very nice message explaining what happened. But even the best applications have bugs and before publishing a Zope 3 application, one should probably provide a more polite message for programming errors. For development, the "Debug" skin contains a nice view for `IException` that shows the exception class and value as well as the traceback.

User and application errors, on the other hand, have often very fancy and elaborate views. User errors commonly implement `IUserError` defined in `zope.app.exceptions.interfaces`. Simple examples of a user error is the message shown when you forgot to enter a name when adding a new content type, like an image. A very good example of an application error is `Unauthorized`, which is raised if a user is not allow to access a particular resource. It's view actually raises an HTTP challenge, so that your browser will ask you for a username and password.

Overall, you should be very careful about classifying your exceptions to which ones are legitimate to reach the users and which aren't. In this chapter, we will create an exception that is raised when a payment is required to access a certain page. We will test the payment exception view by writing a small page that raises the exception.

32.2 Step I: Creating the Exception

Before we get started, create a new package called `exceptionview` in `ZOPE3/src/book`. Then create a file called `interfaces.py` and add the following exception interface and class.

```
1  from zope.interface import implements
2  from zope.interface.common.interfaces import IException
3
4  class IPaymentException(IException):
5      """This is an exception that can be raised by my application."""
6
7  class PaymentException(Exception):
8      implements(IPaymentException)
9
10     # We really do nothing here.
```

▷ Line 2: The interfaces for all common exceptions are defined in `zope.interface.common.interfaces`.

▷ Line 4: You should always inherit `IException` in any exception interface.

▷ Line 7: You should also always inherit `Exception` for any self-written exception. Note that exceptions are considered to be part of a package's API and are therefore always implemented in the `interfaces` module.

32.3 Step II: Providing an Exception View

Now that we have a payment exception, we just have to provide a view for it. However, when the exception occurs, we do not want to return the HTTP status code 200. Instead, we want the status to be 402, which is the "Payment Required" status. In a new module named `browser.py` add the following view class:

```
1 class PaymentExceptionView(object):
2     """This is a view for `IPaymentException` exceptions."""
3
4     def __call__(self, *args, **kw):
5         self.request.response.setStatus(402)
6         return self.index(*args, **kw)
```

▷ Line 4: Since this view will be template-based, the `__call__()` method is usually used to render the template.

▷ Line 5: However, before executing the template, we set the HTTP return status to 402.

▷ Line 6: Now render the template, which is always available under the attribute `index`.

Now we just need a template to render the view. Add the following ZPT code in a file named `error.pt`:

```
1 <html metal:use-macro="context/@@standard_macros/dialog">
2   <body>
3     <div metal:fill-slot="body">
4
5       <h1>402 - Payment Required</h1>
6
7       <p>Before you can use this feature of the site, you have to make a
8          payment to Stephan Richter.</p>
9
10    </div>
11  </body>
12 </html>
```

There is nothing interesting going on in the template, since it has no dynamic components. In `configure.zcml` you can register the page now using

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

  <page
      name="index.html"
      template="error.pt"
      for=".interfaces.IPaymentException"
      class=".browser.PaymentExceptionView"
      permission="zope.Public"
      />

</configure>
```

To register the new package, add a file named `exceptionview-configure.zcml` to `package-includes` having the line:

```
<include package="book.exceptionview" />
```

You can now restart Zope 3. But how can we test whether the view works? There exists currently no code that raises the exception.

32.4 Step III: Testing the Exception View

The easiest way to raise the exception is to write a simple view that just does it. Something like this:

```
from book.exceptionview.interfaces import PaymentException

class RaiseExceptionView(object):
    """The view that raises the exception"""

    def raisePaymentException(self):
        raise PaymentException, 'You are required to pay.'
```

Let's now register the class method as a view on a folder:

```
<page
    name="raiseError.html"
    for="zope.app.folder.interfaces.IFolder"
    class=".browser.RaiseExceptionView"
    attribute="raisePaymentException"
    permission="zope.View"
    />
```

Restart Zope now and enter the URL `http://localhost:8080/raiseError.html` in your browser. You should now see the "Payment Required" exception view.

In your console you should see the following output:

```
----
04-08-23T12:38:01 ERROR SiteError http://localhost:8080/raiseError.html
aceback (most recent call last):
.
ymentException: You are required to pay.
```

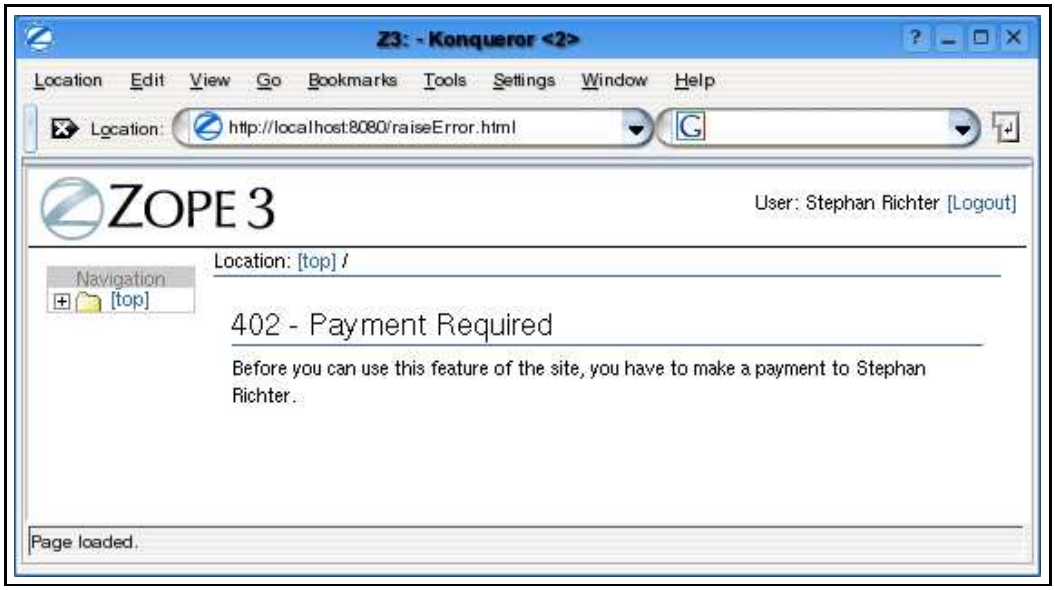


Figure 32.1: This is the view for the `PaymentException` error.

The little experiment we just did is also quickly casted into a functional test. In a new file called `ftests.py` add the following test code:

```
1  import unittest
2
3  from zope.app.tests.functional import BrowserTestCase
4
5  class Test(BrowserTestCase):
6
7      def test_PaymentErrorView(self):
8          response = self.publish("/raiseError.html", handle_errors=True)
9
10         self.assertEqual(response.getStatus(), 402)
11         body = response.getBody()
12         self.assert_('402 - Payment Required' in body)
13         self.assert_('payment to Stephan Richter' in body)
14
15 def test_suite():
16     return unittest.TestSuite((
17         unittest.makeSuite(Test),
18         ))
19
20 if __name__=='__main__':
21     unittest.main(defaultTest='test_suite')
```

▷ Line 8: Make sure that `handle_errors` is set to true, otherwise the publication of this URL will raise the `PaymentException`, failing the test.

▷ Line 10: Using the response's `getStatus()` method we can even get to the HTTP return status, which should be 402, of course. Note that this was not as easily testable using the browser.

▷ Line 11–13: Make sure the page contains the right contents.

If you are not familiar with functional tests, please read the corresponding chapter. You can verify the test by executing

```
python test.py -vpf --dir src/book/exceptionview
```

from your `ZOPE3` directory.

PART VI

Advanced Topics

Not everything you ever want to develop are components that you would allow the user to add and manipulate. This section contains a collection of chapters that deal mainly with the packages outside of `zope.app`. These packages are often useful outside of Zope 3 as well.

Chapter 33: Writing new ZCML Directives

Here we discusses how new directives can be added to a ZCML namespace using meta-directives and/or how to create a new namespace from scratch.

Chapter 34: Implementing a TALES Namespaces

In Zope 3, Zope Page Templates (TALES expressions) can contain namespaces to provide easier access to an object's data and meta-data. While Zope 3 provides a `zope` namespace, it is sometimes extremely helpful to develop your own to expose your product-specific API.

Chapter 35: Changing Traversal Behavior

Similar to Zope 2, one can change the traversal (lookup) behavior for an object, except that this functionality is much more flexible in Zope 3.

Chapter 36: Registering new WebDAV Namespaces

WebDAV is allowed to store and request any namespace on any resource. However, we want to have some control over the namespaces and their values. This chapter explains how to bind Zope 3 attributes and annotations to WebDAV namespaces.

Chapter 37: Using TALES outside of Page Templates

TALES is a powerful expression mechanism that certainly does not only find usage in Page Templates. This chapter tells you how to incorporate TALES into your own Python applications and scripts.

Chapter 38: Developing a new TALES expression

While TALES is powerful in itself, one can make it even more powerful by implementing custom expressions. This chapter will explain step by step how the `sql` expression was created.

Chapter 39: Spacesuits – Objects in Hostile Environements

While the term "spacesuits" is not used in the Zope 3 terminology anymore, it best describes the approach of the new security model, which will be introduced in this chapter.

Chapter 40: The Life of a Request

This chapter will show the request's exciting journey through the server, publisher and publication frameworks.

CHAPTER 33

WRITING NEW ZCML DIRECTIVES

Difficulty

Sprinter

Skills

- Be familiar using ZCML. If necessary, you should read the "Introduction to ZCML" chapter.
- Have a purpose in mind for creating or extending a namespace.

Problem/Task

As you know by now, we use ZCML to configure the Zope 3 framework, especially for globally-available components. When developing complex applications, it is sometimes very useful to develop new and custom ZCML directives to reduce repetitive tasks or simply make something configurable that would otherwise require Python code manipulation. This chapter will implement a small `browser:redirect` directive that defines a view that simply redirects to another URL.

Solution

33.1 Introduction

One of the major design goals of ZCML was to make it very easy for a developer to create a new ZCML directives. We differentiate between simple and complex

directives. Simple directives consist of one XML element that causes a set of actions. Complex directives are wrapper-like directives that can contain other sub-directives. They usually do not cause actions themselves, but provide data that is applicable to most of the sub-directives. In this chapter, however, we will just create a simple directive; the complex ones are not much more difficult.

There are three simple steps to implementing a directive. First, you develop the directive's schema, which describes the attributes the XML element can and must have. Like with any schema, you can specify, whether an attribute is required or not. When the XML is parsed, the unicode values that are returned from the parser are automatically converted to Python values as described by the field. Besides the common fields, such as `TextLine` or `Int`, you also have special configuration fields, such as `GlobalObject`, which automatically converts a Python reference to a Python object. A full list of additional fields is provided in the "Introduction to ZCML" chapter. Directive schemas are commonly placed in a file called `metadirectives.py`.

The second step is to develop a handler for the directive, which is for simple directives a function taking the attributes as arguments. The first attribute of the handler is the `context` of the configuration. If you have a complex directive, the handler is usually a class, where the constructor takes the attributes of the directive as arguments. Each sub-directive is then a method on the class. The class must also be callable, so that it can be called when the complex directive is closed. It is very important to note, that the directives should *not perform any actions*, but only declare the actions as we will see later. This way the configuration mechanism can detect configuration conflict. By convention the handlers are stored in `metaconfigure.zcml`.

Once the directive schema and handler are written, we can now register the ZCML directive using the ZCML `meta` namespace, which is usually done in a configuration file named `meta.zcml`. The meta-configuration file is then registered in `packages-includes` using a filename like ⟨`package`⟩`-meta.zcml`.

Now that you have an overview over the necessary tasks, let's get our hands dirty. As mentioned before the goal is to provide a directive that creates a view that makes a simple redirect. A view must always be defined for a particular object and needs a name to be accessible under. We should also optionally allow a layer to be specified. Usually, we also want to specify a permission, but since this view just redirects to another we simply make the view public. The final attribute we need for the directive is the `url`, which specifies the URL we want to direct to. so, first create a package named `redirect` in `ZOPE3/src/book/` (and don't forget about `__init__.py`).

33.2 Step I: Developing the Directive Schema

In a new file named `metadirectives.py` add the following schema.

```
1  from zope.interface import Interface
2  from zope.configuration.fields import GlobalObject
3  from zope.schema import TextLine
4
5  class IRedirectDirective(Interface):
6      """Redirects clients to a specified URL."""
7
8      name = TextLine(
9          title=u"Name",
10         description=u"The name of the requested view.")
11
12     for_ = GlobalObject(
13         title=u"For Interface",
14         description=u"The interface the directive is used for.",
15         required=False)
16
17     url = TextLine(
18         title=u"URL",
19         description=u"The URL the client should be redirected to.")
20
21     layer = TextLine(
22         title=u"Layer",
23         description=u"The layer the redirect is defined in.",
24         required=False)
```

- ▷ Line 2: As you can see, all configuration-specific fields, like `GlobalObject` are defined in `zope.configuration.fields`.
- ▷ Line 3: However, you can also use any of the conventional fields as well.
- ▷ Line 5: The directive schemas are just schemas like any other ones. There is no special base-class required.
- ▷ Line 12: Whenever you have an attribute whose name is a Python keyword, then simply add an underscore behind it; the underscore will be safely ignored during runtime.

33.3 Step II: Implementing the Directive Handler

The handler is added to `metaconfigure.zcml`.

```
1  from zope.app.publisher.browser.viewmeta import page
2
3  class Redirect(object):
4      """Redirects to a specified URL."""
5      url = None
6
7      def __call__(self):
```

```
8         self.request.response.redirect(self.url)
9
10
11 def redirect(_context, name, url, for_=None, layer='default'):
12     # define the class that performs the redirect
13     redirectClass = type(str("Redirect %s for %s to '%s'" %(name, for_, url)),
14         (Redirect,), {'url' : url})
15
16     page(_context, name, 'zope.Public', for_, layer, class_=redirectClass)
```

▷ Line 1: Since we are just defining a new page, why not reuse the page-directive handler? This makes the implementation of our handler much simpler.

▷ Line 3–8: This is the base view class. We simply allow a URL to be set on it. When the view is called, we simply redirect the HTTP request. There is no need to implement `IBrowserPublisher` or `IBrowserView` here, since the `page()` function will mix-in all of these APIs plus implementation.

▷ Line 11–16: This is the actual handler of the directive. The first step is to create a customized version of the view class by merging in the URL (line 13). Then we simply call the page-directive handler, where we use the public permission. The page-directive handler hides a lot of the glory details of defining a full-blown view, including creating configuration actions.

An action is created by calling `_context.action`)). This function supports the following arguments:

- `discriminator` – This is a unique identifier that is used to recognize a particular action. It is very important that no two actions have the same discriminator when starting Zope. This allows us to use the discriminator for conflict resolution and spotting duplicate actions. It is usually a tuple.
- `callable` – Here we specify the callable (usually a method or function) that is called when the action is executed.
- `args` & `kw` – Arguments and keywords that are passed to the callable as arguments on execution time.

33.4 Step III: Writing the Meta-Configuration

Now that we have all pieces of the directive, let's register it in `meta.zcml`.

```
1 <configure
2     xmlns="http://namespaces.zope.org/meta">
3
4   <directives namespace="http://namespaces.zope.org/browser">
5     <directive
6       name="redirect"
```

```
7       schema=".metadirectives.IRedirectDirective"
8       handler=".metaconfigure.redirect"
9       />
10    </directives>
11
12  </configure>
```

- ▷ Line 2: The `meta` namespace is used to define new ZCML directives.
- ▷ Line 4 & 10: The `meta:directives` directive is used to specify the namespace under which the directives will be available. In our case it is the `browser` namespace.
- ▷ Line 5–9: The `meta:directive` directive is used to register a simple directive. The `name` is the name as which the directive will be known/accessible. The `schema` specified the directive schema and the `handler` the directive handler, both of which we developed before.

  If you develop a complex directive, you would use the `meta:complexDirective` directive, which supports the same attributes. Inside a complex directive you can then place `meta:subdirective` directives, which define the sub-directives of the complex directive. You might want to look into the "Registries with Global Utilities" chapter for an example of a relatively simple complex directive.

You now have to register the new directive with the Zope 3 system by placing a file named `redirect-meta.zcml` in `package-includes`. It should have the following content:

```
1 <include package="book.redirect" file="meta.zcml" />
```

The next time you restart Zope, the directive should be available.

33.5 Step IV: Testing the Directive

The best way to test the directive is to use it. Let's have the view "manage.html" be redirected to "manage" for all folders. In a new configuration file, `configure.zcml`, add the following directives:

```
1 <configure
2     xmlns="http://namespaces.zope.org/browser">
3
4   <redirect
5       name="manage.html"
6       for="zope.app.folder.interfaces.IFolder"
7       url="manage" />
8
9 </configure>
```

▷ Line 2: As specified, the `redirect` directive is available via the `browser` namespace.

▷ Line 5: The name is the view that must be called to initiate the redirection.

▷ Line 6: The redirection will only be available for folders.

▷ Line 7: The target URL is the relative name "manage".

After adding `redirect-configure.zcml` containing

```
1 <include package="book.redirect" />
```

to `package-includes`, restart Zope 3. You should now be able to call the URL `http://localhost:8080/@@manage.html`, which should bring you to `http://localhost:8080/@@contents.html`, since "manage" just redirects to "contents.html".

This functionality can be easily duplicated in a functional test. Put the following test case into a file named `ftests.py`:

```
1  import unittest
2
3  from zope.app.tests.functional import BrowserTestCase
4
5
6  class Test(BrowserTestCase):
7
8      def test_RedirectManageHtml(self):
9          response = self.publish("/manage.html")
10
11         self.assertEqual(response.getStatus(), 302)
12         self.assertEqual(response.getHeader('Location'), 'manage')
13
14
15 def test_suite():
16     return unittest.makeSuite(Test)
17
18 if __name__=='__main__':
19     unittest.main(defaultTest='test_suite')
```

If you are not familiar with the `BrowserTestCase` API, I suggest you read the "Writing Functional Tests" chapter. Otherwise the test is straightforward and you can execute in the usual manner.

CHAPTER 34

IMPLEMENTING A TALES NAMESPACES

Difficulty

Newcomer

Skills

- Be familiar with TAL and TALES (in the context of Page Templates).
- You should feel comfortable with ZCML.

Problem/Task

Zope 3 exposes a fair amount of its API in TAL and TALES through expression types (path, python, string and sql [add-on]) as well as the TALES namespaces, such as `zope`. However, sometimes this is not powerful enough or still requires a lot of Python view class coding. For this reason Zope 3 allows you to add new TALES namespace. This chapter will demonstrate on how to write and register a new TALES namespace.

Solution

TALES namespaces use path adapters to implement the adaptation. Path adapters are used to adapt an object to another interface while traversing a path. The name of the adapter is used in the path to select the correct adapter. An example is the `zope` TALES namespace.

```
<p tal:content="context/zope:modified" />
```

In this example, the object is adapted to the `IZopeTalesAPI`, which provides many convenience methods, including the exposure of the Dublin Core. Then the `modified()` method is called on the adapter, which returns the modification date of the context object.

While the standard `zope` TALES namespace deals with retrieving additional data about the component, it does not handle the output format of the data. So it would be a good idea to specify a new namespace that deals exclusively with the formatting of objects. To keep this chapter short, we will only concentrate on the `full` display of dates, times and datetimes. The user's locale functions are used to do the actual formatting, so that the developer's effort will be minimal and the output is correctly localized.

The code for this example is located in `book/formatns`. Therefore, create this product and do not forget the `__init__.py` file in the directory.

34.1 Step I: Defining the Namespace Interface

Let's start out by defining the interface of the namespace. The interface specifies all the functions that will be available in the namespace. Note that the code available in the repository has a lot more functions, but since the code is fairly repetitive, I decided not to put it all into the text.

```
from zope.interface import Interface

class IFormatTalesAPI(Interface):

    def fullDate(self):
        """Returns the full date using the user's locale.

        The context of this namespace must be a datetime object,
        otherwise an exception is raised.
        """

    def fullTime(self):
        """Returns the full time using the user's locale.

        The context of this namespace must be a datetime object,
        otherwise an exception is raised.
        """

    def fullDateTime(self):
        """Returns the full datetime using the user's locale.

        The context of this namespace must be a datetime object,
        otherwise an exception is raised.
        """
```

While every TALES namespace also has to implement `ITALESFunctionNamespace`, we do not inherit from this interface here, but simply merge it in in the implementation. This has the advantage that the `IFormatTalesAPI` interface can be reused elsewhere.

34.2 Step II: Implementing the Namespace

The actual code of the namespace is not much harder than the interface, if you have played with the user locales before. Add the following implementation in the package's `__init__.py` file.

```
1  from zope.interface import implements
2  from zope.tales.interfaces import ITALESFunctionNamespace
3  from zope.security.proxy import removeSecurityProxy
4  from interfaces import IFormatTalesAPI
5
6
7  class FormatTalesAPI(object):
8
9      implements(IFormatTalesAPI, ITALESFunctionNamespace)
10
11     def __init__(self, context):
12         self.context = context
13
14     def setEngine(self, engine):
15         """See zope.tales.interfaces.ITALESFunctionNamespace"""
16         self.locale = removeSecurityProxy(engine.vars['request']).locale
17
18     def fullDate(self):
19         """See book.formatns.interfaces.IFormatTalesAPI"""
20         return self.locale.dates.getFormatter(
21             'date', 'full').format(self.context)
22
23     def fullTime(self):
24         """See book.formatns.interfaces.IFormatTalesAPI"""
25         return self.locale.dates.getFormatter(
26             'time', 'full').format(self.context)
27
28     def fullDateTime(self):
29         """See book.formatns.interfaces.IFormatTalesAPI"""
30         return self.locale.dates.getFormatter(
31             'dateTime', 'full').format(self.context)
```

▷ Line 2: Here you see where to import the `ITALESFunctionNamespace` interface from. Note that this interface is not totally necessary, but without it, the engine will not be set on the namespace, which makes it impossible to reach the request.

▷ Line 11–12: All TALES function namespaces must be implemented as adapters. The object they adapt is the object that was evaluated from the previous part of the path expression.

▷ Line 14–16: This method implements the only requirement the `ITALESFunctionNamespace` interface poses. This method provides the some additional context about the user and the entire request. For this namespace, however, we are only interested in the locale, so we get it. Interestingly enough, the engine is security-wrapped, so that the request is automatically security-wrapped, which is unusal and therefore no security declaration exists for accessing the `locale` attribute. Therefore we have to remove all the security proxies from the `request` before accessing the `locale` object. Note that this is safe, since all of the namespace functions only read data from the locale, but do not do any write operations.

▷ Line 18–31: Here you can see the trivial implementations of the namespace functions. The `locale` object provides all the functionality we need. From the locale itself we can retrieve the date, time or datetime formatting objects using `locale.dates.getFormat()`. The method expect the name of the format to be specified. The four names supported by ICU (on which the locale support is based) are "short", "medium", "long", and "full". The formatting objects have a method called `format()`, which converts the datetime object into a localized string representation.

That was easy, wasn't it? Next we are going to test the functionality.

34.3 Step III: Testing the Namespace

Here we are only going to test whether the namespace works by itself; we are not checking whether the namespace will work correctly in TALES, since this should be tested in the TALES implementation. The tricky part of the test is to create a sufficient `Engine` object, so that the code can access the `request`:

```
1  import unittest
2  from datetime import datetime
3  from zope.publisher.browser import TestRequest
4  from zope.testing.doctestunit import DocTestSuite
5  from book.formatns import FormatTalesAPI
6
7  class Engine:
8      vars = {'request': TestRequest(environ={'HTTP_ACCEPT_LANGUAGE': 'en'})}
9
10 def getFormatNamespace(context):
11     ns = FormatTalesAPI(context)
12     ns.setEngine(Engine())
13     return ns
14
15 def fullDate():
16     """
17     >>> ns = getFormatNamespace(datetime(2003, 9, 16, 16, 51, 01))
18     >>> ns.fullDate()
19     u'Tuesday, September 16, 2003'
```

```
20      """
21
22  def fullTime():
23      """
24      >>> ns = getFormatNamespace(datetime(2003, 9, 16, 16, 51, 01))
25      >>> ns.shortTime()
26      u'4:51:01 PM +000'
27      """
28
29  def fullDateTime():
30      """
31      >>> ns = getFormatNamespace(datetime(2003, 9, 16, 16, 51, 01))
32      >>> ns.fullDateTime()
33      u'Tuesday, September 16, 2003 4:51:01 PM +000'
34      """
35
36  def test_suite():
37      return DocTestSuite()
38
39  if __name__ == '__main__':
40      unittest.main(defaultTest='test_suite')
```

- Line 7–8: This is the most minimal stub implementation of the engine, which sufficiently provides the request, as it is required by the namespace.
- Line 10–13: This helper function creates the namespace instance for us and also sets the engine, so that we are ready to use the returned namespace object.
- Line 15–34: These are the tests in the Zope "Doc Unit Test" format, which is easy to read. You see what's going on. The code inside the doc string is executed and it is checked whether the returned value equals the expected value.
- Line 36–40: This is the usual unit doctest boilerplate. The only difference is that we create a `DocTestSuite` without passing in the module name. If no name is specified to the `DocTestSuite` constructor, the current module is searched for tests.

You can run the tests as usual, of course, using

```
python test.py -vpu --dir src/book/formatns
```

34.4 Step IV: Wiring the Namspace into Zope 3

The last task is to hook up the namespace to the rest of the framework. This is simply done by registering a normal adapter that provides the `IPathAdapter` interface. Place the following code into the "configure.zcml" file:

```
1  <configure
2      xmlns="http://namespaces.zope.org/zope">
3
4    <adapter
5        for="*"
6        provides="zope.app.traversing.interfaces.IPathAdapter"
7        factory=".FormatTalesAPI"
8        name="format" />
9
10 </configure>
```

▷ Line 5: Register this namespace as an adapter for all possible objects, even though in our case we could restrict it to `IDateTime` instances. However, the idea is that the `format` namespace will support many different object types.

▷ Line 8: The name of the adapter is extremly important, since it is the name as which the adapter will be available in the path expressions.

Now hook the configuration into Zope 3 by adding a file called `formatns-configure.zcml` to `package-includes` having the following line as content:

```
1 <include package="book.formatns" />
```

34.5 Step VI: Trying the `format` Namespace

First you need to restart Zope. Then create a "ZPT Page" and add the following content:

```
1 <html>
2   <body>
3     <h1 tal:content="context/zope:modified/format:fullDateTime">
4       Tuesday, September 16, 2003 04:51:01 PM +000
5     </h1>
6   </body>
7 </html>
```

▷ Line 3–4: This line displays the modification datetime of the root folder as a "full" date time. The output is even localized to the user's preferred language and format.

A great aspect of the function namespace concept is that several namespace calls can be piped together. In the example above you can see how the `zope` namespace extracts the modification datetime of the root folder, and this datetime object is then passed to the `format` namespace to create the localized human-readable representation.

CHAPTER 35

CHANGING TRAVERSAL BEHAVIOR

Difficulty

Sprinter

Skills

- Be familiar with the Zope 3 component architecture and testing framework.
- Some knowledge about the container interfaces is helpful. Optional.

Problem/Task

Zope 3 uses a mechanism called "traversal" to resolve an object path, as given by a URL, to the actual object. Obviously, there is some policy involved in the traversal process, as objects must be found, namespaces must be resolved, and even components, such as views, be looked up in the component architecture. This also means that these policies can be changed and replaced. This chapter will show you how to change the traversal policy, so that the container items are not case-sensitive anymore.

Solution

In Zope 3, traversers, objects that are responsible for using a path segment to get from one object to another, are just implemented as views of the respective presentation type. In principle the traverser only has to implement `IPublishTraverse` (located in `zope.publisher.interfaces`), which specifies a method named `publishTraverse(request,name)` that returns the traversed object. The browser implementation, for example, is simply a view that tries to

resolve `name` using its `context`. Whether the method tries to access sub-objects or look up views called `name` is up to the specific implementation, like the `zope.app.container.traversal.ContainerTraverser` for the browser.

35.1 Step I: The Case-Insensitive Folder

As mentioned before, in this chapter we are going to implement a case-insensitive traverser and a sample folder that uses this traverser called `CaseInsensitiveFolder`. Let's develop the latter component first. All we need for the case-insensitive folder is an interface and a factory that provides the for a normal `Folder` instance.

Create a package called `insensitivefolder` in the `book` package. In the `__init__.py` file add the following interface and factory:

```
1  from zope.component.interfaces import IFactory
2  from zope.app.folder import Folder
3  from zope.app.folder.interfaces import IFolder
4  from zope.interface import implements, implementedBy
5  from zope.interface import directlyProvides, directlyProvidedBy
6
7  class ICaseInsensitiveFolder(IFolder):
8      """Marker for folders whose contained items keys are case insensitive.
9
10     When traversing in this folder, all names will be converted to lower
11     case. For example, if the traverser requests an item called 'Foo', in
12     reality the first item matching 'foo' or any upper-and-lowercase
13     variants are looked up in the container."""
14
15 class CaseInsensitiveFolderFactory(object):
16     """A Factory that creates case-insensitive Folders."""
17     implements(IFactory)
18
19     def __call__(self):
20         """See zope.component.interfaces.IFactory
21
22         Create a folder and mark it as case insensitive.
23         """
24         folder = Folder()
25         directlyProvides(folder, directlyProvidedBy(folder),
26                          ICaseInsensitiveFolder)
27         return folder
28
29     def getInterfaces(self):
30         """See zope.component.interfaces.IFactory"""
31         return implementedBy(Folder) + ICaseInsensitiveFolder
32
33 caseInsensitiveFolderFactory = CaseInsensitiveFolderFactory()
```

Instead of developing a new content type, we create a factory that tags on the marker interface making the `Folder` instance an `ICaseInsensitiveFolder`. This is a classic example of declaring and using a factory directly. Factories are used in many places, but usually they are auto-generated in ZCML handlers.

The factory is simply registered in ZCML using

```
1  <configure
2      xmlns="http://namespaces.zope.org/zope"
3      xmlns:browser="http://namespaces.zope.org/browser"
4      i18n_domain="zope">
5
6  <factory
7      id="zope.CaseInsensitiveFolder"
8      component=".caseInsensitiveFolderFactory"
9      />
10
11 <browser:addMenuItem
12     factory="zope.CaseInsensitiveFolder"
13     title="Case insensitive Folder"
14     description="A simple case insensitive Folder."
15     permission="zope.ManageContent"
16     />
17
18 <browser:icon
19     name="zmi_icon"
20     for=".ICaseInsensitiveFolder"
21     file="cifolder_icon.png"
22     />
23
24 </configure>
```

- ▷ Line 6–9: Declare the factory. The `id` must be a valid `Id` field value.
- ▷ Line 11–16: Declare an add menu item entry using the factory id, as specified in the `id` attribute before.
- ▷ Line 18–22: Register also a custom icon for the case-insensitive folder, so that we can differentiate it from the other folders. The icon can be found in the repository.

35.2 Step II: The Traverser

Now we have a new content type, but it does not do anything special. We have to implement the special traverser for the case-insensitive folder. Luckily, we do not have to implement a new container traverser from scratch, but just use the standard `ContainerTraverser` and replace the `publishTraverse()` method to be a bit more flexible and ignore the case of the item names. In the `__init__.py` file add the following traverser class:

```
1  from zope.publisher.interfaces import NotFound
2
3  from zope.app import zapi
4  from zope.app.container.traversal import ContainerTraverser
5
6  class CaseInsensitiveFolderTraverser(ContainerTraverser):
7
8      __used_for__ = ICaseInsensitiveFolder
```

```
9
10      def publishTraverse(self, request, name):
11          """See zope.publisher.interfaces.browser.IBrowserPublisher"""
12          subob = self._guessTraverse(name)
13          if subob is None:
14              view = zapi.queryView(self.context, name, request)
15              if view is not None:
16                  return view
17
18              raise NotFound(self.context, name, request)
19
20          return subob
21
22      def _guessTraverse(self, name):
23          for key in self.context.keys():
24              if key.lower() == name.lower():
25                  return self.context[key]
26          return None
```

▷ Line 8: Just as information, this traverser is only meant to be used with `ICaseInsensitiveFolder` components. However, the following code is generic enough that it would work with any object implementing `IReadContainer`. Note that the most generic container traverser is registered for `ISimpleReadContainer`, which is not sufficient here, since we make use of the `keys()` method, which is not available in `ISimpleReadContainer`.

▷ Line 10–20: First we try to find the name using the private `_guessTraverse()` method. If no object was found in the items of the container, we check whether `name` could be a view and return it. If the `name` does not point to an item or a view, then we need to raise a `NotFound` error.

Note that the implementation of this method could have been more efficient. We could first try to get the object using `_guessTraverse()` and upon failure forward the request to the original `publishTraverse()` method of the base class. Then the code would look like this:

```
def publishTraverse(self, request, name):
    subob = self._guessTraverse(name)
    if subob is not None:
        return subob
    return super(CaseInsensitiveFolderTraverser,
                 self).publishTraverse(request, name)
```

However, this would have hidden some of the insights on how `publishTraverse()` should behave.

▷ Line 22–26: Here we try to look up the name without caring about the case. This works both ways. The keys of the container *and* the provided `name` are converted to all lower case. We then compare the two. If a match is found, the value for

the key is returned. Note that we need to keep the original key (having upper and lower case), since the container still manages the keys in a case-sensitive manner.

The traverser is registered via ZCML simply using the `zope:view` directive:

```
1 <view
2     for=".ICaseInsensitiveFolder"
3     type="zope.publisher.interfaces.browser.IBrowserRequest"
4     factory=".CaseInsensitiveFolderTraverser"
5     provides="zope.publisher.interfaces.browser.IBrowserPublisher"
6     permission="zope.Public"
7     />
```

▷ Line 2: Register the view only for case-insensitive folders.

▷ Line 3: Make sure that this traverser is only used for browser requests.

▷ Line 4: It is very important to specify the provided interface here, so that we know that the object is a browser publisher and implements the sufficient interfaces for traversal.

▷ Line 5: We want to allow everyone to be able to traverse through the folder, since it does not you anyone any special access. All methods of the returned object are protected separately anyways.

To register the product with Zope 3, add a file named `insensitivefolder-configure.zcml` to `package-includes`. It should contain the following line:

```
1 <include package="book.insensitivefolder" />
```

Once you restart Zope 3, the new folder should be available to you.

35.3 Step III: Unit Tests

Unit tests can be quickly written, since we can make use of the original container traverser's unit tests and setup. Open a `tests.py` file to insert the following test code.

```
1  import unittest
2  from zope.app.container.tests import test_containertraverser
3  from book.insensitivefolder import CaseInsensitiveFolderTraverser
4
5  class Container(test_containertraverser.TestContainer):
6
7      def keys(self):
8          return self.__dict__.keys()
9
10     def __getitem__(self, name):
11         return self.__dict__[name]
12
```

```
13 class InsensitiveCaseTraverserTest(test_containertraverser.TraverserTest):
14
15     def _getTraverser(self, context, request):
16         return CaseInsensitiveFolderTraverser(context, request)
17
18     def _getContainer(self, **kw):
19         return Container(**kw)
20
21     def test_allLowerCaseItemTraversal(self):
22         self.assertEquals(
23             self.traverser.publishTraverse(self.request, 'foo'),
24             self.foo)
25         self.assertEquals(
26             self.traverser.publishTraverse(self.request, 'foO'),
27             self.foo)
28
29 def test_suite():
30     return unittest.TestSuite((
31         unittest.makeSuite(InsensitiveCaseTraverserTest),
32         ))
33
34 if __name__ == '__main__':
35     unittest.main(defaultTest='test_suite')
```

▷ Line 7–11: The original test container has to be extended to support `keys()` and `__getitem__()`, since both of these methods are need for the case-insensitive traverser.

▷ Line 15–16: A setup helper method that returns the traverser. This method is used in the `setUp()` method to construct the environment.

▷ Line 18–19: Another helper method that allows us to specify a custom container. This method is used in the `setUp()` method to construct the environment.

▷ Line 21–27: Most of the functionality is already tested in the base test case. Here we just test that the case of the letters is truly ignored.

▷ Line 29–35: This is the usual unit test boilerplate.

As always, the tests are directly executable, once Zope 3 is in your path.

35.4 Step IV: Functional Tests

Before we let the code be run in the wild, we should test it some more in a fairly contained environment. The following functional test is pretty straight forward and mimics the unit tests.

Open a file called `ftests.py` and add

```
import unittest
from zope.app.tests.functional import BrowserTestCase
from zope.publisher.interfaces import NotFound

class TestCaseInsensitiveFolder(BrowserTestCase):

    def testAddCaseInsensitiveFolder(self):
        # Step 1: add the case insensitive folder
        response = self.publish(
            '/+/action.html',
            basic='mgr:mgrpw',
            form={'type_name': book.CaseInsensitiveFolder',
                  'id': u'cisf'})
        self.assertEqual(response.getStatus(), 302)
        self.assertEqual(response.getHeader('Location'),
                         'http://localhost/@@contents.html')
        # Step 2: add the file
        response = self.publish('/cisf/+/action.html',
                                basic='mgr:mgrpw',
                                form={'type_name': u'zope.app.content.File',
                                      'id': u'foo'})
        self.assertEqual(response.getStatus(), 302)
        self.assertEqual(response.getHeader('Location'),
                         'http://localhost/cisf/@@contents.html')
        # Step 3: check that the file is traversed
        response = self.publish('/cisf/foo')
        self.assertEqual(response.getStatus(), 200)
        response = self.publish('/cisf/foO')
        self.assertEqual(response.getStatus(), 200)
        self.assertRaises(NotFound, self.publish, '/cisf/bar')


def test_suite():
    return unittest.TestSuite((
        unittest.makeSuite(TestCaseInsensitiveFolder),
        ))

if __name__ == '__main__':
    unittest.main(defaultTest='test_suite')
```

There is really nothing interesting about this test. If you are not familiar with functional tests, read the corresponding chapter in the "Writing Tests" part of the book.

In Zope 2 it was common to change the traversal behavior of objects and container-ish objects. In Zope 3, however, you will not need to implement your own traversers, since most of the time it is better and easier to write a custom `IReadContainer` content component.

The complete code of this product can be found at `book/insensitivefolder`. It was originally written by Vincenzo Di Somma and has been maintained by many developers throughout the development of Zope 3.

Exercises

1. The current implementation will only allow case-insensitive lookups through browser requests. But what about

   (a) XML-RPC and WebDAV, and

   (b) FTP?

   Extend the existing implementations to support these protocols as well.

2. You might have already noticed that the implementation of this traverser is not quiet perfect and it might actually behave differently on various systems. Let's say you have an object called "Foo" and "foo". The way `_guessTraverse()` is implemented, it will use the key that is listed first in the list returned by `keys()`. However, the order the keys returned varies from system to system. Fix the behavior so that the behavior (whatever you decide it to be) is the same on every system. (Hint: This is a two line fix.)

3. It might be desirable, to change the policy of the name lookup a bit by giving exact key matches preference to case-insensitive matches. Change the traverser to implement this new policy.

CHAPTER 36

REGISTERING NEW WEBDAV NAMESPACES

Difficulty

Sprinter

Skills

- You should know Zope 3's component architecture.
- Be familiar with the WebDAV standard as proposed in RFC 2518.

Problem/Task

WebDAV, as an advanced application of HTTP and XML, supports an unlimited amount meta-data to be associated with any resource. This, of course, is non-sense for Zope related applications and could potentially make Zope vulnerable to DoS attacks, since someone could try to add huge amounts of meta-data to a resource. A namespace registry was created that manages the list of all available namespaces per content type (interface). This chapter will show you how to enable a new namespace called `photo` for an `IImage` object.

Solution

36.1 Introduction

As mentioned above, WebDAV's unlimited support for XML namespaces make WebDAV very powerful but also provide an easy target for malicious attacks if not properly controlled. Therefore we would like to control, an object's WebDAV namespaces as well as the permissions required to access and modify the the namespace's attributes. Furthermore, there is a desire to integrate the namespace data and handling into Zope 3 as much as possible, so that other components could easily reuse the information.

First of all, namespaces with attributes are really just schemas, so that we are able to describe a namespace using the Zope 3 `zope.schema` package. Now we are even able to write WebDAV widgets for the schema field types. Adapters are used to connect a WebDAV namespace to a content type or any other object. Using schemas, widgets and adapters we are able to completely describe the namespace and the storage of the data.

The last step of the process is to register the schema as a WebDAV namespace. This is done by registering the schema as a `IDAVNamespace`, where the name of the utility is the WebDAV namespace URI. However, the `dav` ZCML namespace provides a nice directive, `provideInterface`, which registers the utility for you.

If one wants to provide a new namespace for a given object, the main task for the developer consists of creating a schema for the namespace and to provide an adapter from the object to the schema. The goal of this chapter will be to provide some additional meta-data information about images that have been taken by digital cameras – images that are photos.

Let's create a new package called `photodavns` in ⟨ZOPE3⟩`/src/book`.

36.2 Step I: Creating the Namespace Schema

The schema of the photo should contain some information that are usually provided by the camera. To implement the schema, open a new file `interfaces.py` and add the following code.

```
1  from zope.interface import Interface
2  from zope.schema import Text, TextLine, Int, Float
3
4  photodavns = "http://namespaces.zope.org/dav/photo/1.0"
5
6  class IPhoto(Interface):
7      """A WebDAV namespace to store photo-related meta data.
8
9      The 'IPhoto' schema/namespace can be used in WebDAV clients to determine
10     information about a particular picture. Obviously, this namespace makes
11     only sense on Image objects.
12     """
```

```
13
14      height = Int(
15          title=u"Height",
16          description=u"Specifies the height in pixels.",
17          min=1)
18
19      width = Int(
20          title=u"Width",
21          description=u"Specifies the width in pixels.",
22          min=1)
23
24      equivalent35mm = TextLine(
25          title=u"35mm equivalent",
26          description=u"The photo's size in 35mm is equivalent to this amount")
27
28      aperture = TextLine(
29          title=u"Aperture",
30          description=u"Size of the aperture.")
31
32      exposureTime = Float(
33          title=u"Exposure Time",
34          description=u"Specifies the exposure time in seconds.")
```

▷ Line 4: The name of the namespace is also part of the interface, so declare it here. The name must be a valid URI, otherwise the configuration directive that registers the namespace will fail.

There is nothing more of interest in this code; at this time you should be very comfortable with interfaces and schemas. If not, please read the chapters on interfaces and schemas.

36.3 Step II: Implementing the `IPhoto` to `IImage` Adapter

Next we need to implement the adapter, which will use annotations to store the attribute data. That means that the `IImage` object must also implement `IAttributeAnnotable`. With the knowledge of the annotations chapter, the following implementation should seem simple. Place it in the `__init__.py` file of the package.

```
1  from persistent.dict import PersistentDict
2  from zope.interface import implements
3  from zope.schema import getFieldNames
4  from zope.app.annotation.interfaces import IAnnotations
5  from zope.app.file.interfaces import IImage
6  from interfaces import IPhoto, photodavns
7
8  class ImagePhotoNamespace(object):
9      """Implement IPhoto namespace for IImage."""
10
11     implements(IPhoto)
12     __used_for__ = IImage
```

```
13
14      def __init__(self, context):
15          self.context = context
16          self._annotations = IAnnotations(context)
17          if not self._annotations.get(photodavns):
18              self._annotations[photodavns] = PersistentDict()
19
20      def __getattr__(self, name):
21          if not name in getFieldNames(IPhoto):
22              raise AttributeError, "'%s' object has no attribute '%s'" %(
23                  self.__class__.__name__, name)
24          return self._annotations[photodavns].get(name, None)
25
26      def __setattr__(self, name, value):
27          if not name in getFieldNames(IPhoto):
28              return super(ImagePhotoNamespace, self).__setattr__(name, value)
29          field = IPhoto[name]
30          field.validate(value)
31          self._annotations[photodavns][name] = value
```

▷ Line 14–18: During initialization, get the annotations for the `IImage` object and create a dictionary where all the attribute values will be stored. Make sure that the dictionary is a `PersistentDict` instance, since otherwise the data will not be stored permanently in the ZODB.

▷ Line 20–24: If the name of the requested attribute corresponds to a field in `IPhoto` then we get the value from the annotations otherwise fail with an attribute error.

▷ Line 26–31: We want to set attributes differently, if they are fields in the `IPhoto` schema. If the name is a field, then the first task is to get the field which is then used to validate the value. This way we can enforce all specifications provided for the fields in the schema. If the validation passes, then store the value in the annotations.

36.4 Step III: Unit-Testing and Configuration

For the unit tests of the adapter, we use doc tests. So we extend the adapter's class doc string to become:

```
"""Implement IPhoto namespace for IImage.

Examples:

>>> from zope.app.file.image import Image
>>> image = Image()
>>> photo = IPhoto(image)

>>> photo.height is None
True
>>> photo.height = 768
```

```
12 >>> photo.height
13 768
14 >>> photo.height = u'100'
15 Traceback (most recent call last):
16 ...
17 WrongType: (u'100', (<type 'int'>, <type 'long'>))
18
19 >>> photo.width is None
20 True
21 >>> photo.width = 1024
22 >>> photo.width
23 1024
24
25 >>> photo.equivalent35mm is None
26 True
27 >>> photo.equivalent35mm = u'41 mm'
28 >>> photo.equivalent35mm
29 u'41 mm'
30
31 >>> photo.aperture is None
32 True
33 >>> photo.aperture = u'f/2.8'
34 >>> photo.aperture
35 u'f/2.8'
36
37 >>> photo.exposureTime is None
38 True
39 >>> photo.exposureTime = 0.031
40 >>> photo.exposureTime
41 0.031
42
43 >>> photo.occasion
44 Traceback (most recent call last):
45 ...
46 AttributeError: 'ImagePhotoNamespace' object has no attribute 'occasion'
47 """
```

You can see that the example code covers pretty much every possible situation.

- ▷ Line 5–7: Use the standard `Image` content component as context for the adapter. Then we use the component architecture to get the adapter. This already tests whether the constructor – which is not trivial in this case – does not cause an exception.
- ▷ Line 14–17: Test that the validation of the field's values works correctly.
- ▷ Line 43–46: We also need to make sure that no non-existing attributes can be assigned a value.

To make the tests runnable, add a file named `tests.py` and add the following test code.

```
1 import unittest
2 from zope.interface import classImplements
3 from zope.testing.doctestunit import DocTestSuite
```

```
4  from zope.app.annotation.interfaces import IAttributeAnnotatable
5  from zope.app.file.image import Image
6  from zope.app.file.interfaces import IImage
7  from zope.app.tests import ztapi, placelesssetup, setup
8  from book.photodavns.interfaces import IPhoto
9  from book.photodavns import ImagePhotoNamespace
10
11 def setUp(test):
12     placelesssetup.setUp()
13     ztapi.provideAdapter(IImage, IPhoto, ImagePhotoNamespace)
14     setup.setUpAnnotations()
15     classImplements(Image, IAttributeAnnotatable)
16
17 def test_suite():
18     return unittest.TestSuite((
19         DocTestSuite('book.photodavns',
20                      setUp=setUp, tearDown=placelesssetup.tearDown),
21         ))
22
23 if __name__ == '__main__':
24     unittest.main(defaultTest='test_suite')
```

▷ Line 13–15: We need to setup some additional adapters to make the tests work. First, of course, we need to register our adapter. Then we also need to provide the `AttributeAdapter`, so that the `ImagePhotoNamespace` will find the annotations for the image. Luckily the `zope.app.tests.setup` module has a convenience function to do that. Finally, since `Image` does not implement `IAttributeAnnotable` directly (it is usually done in a ZCML directive), we need to declare it manually for the unit tests.

▷ 19–20: The `setUp()` and `tearDown()` functions for a doc test can be passed as keyword arguments to the `DocTestSuite` constructor.

From the Zope 3 root directory, you can now execute the tests using

```
python test.py -vpu --dir /src/book/photodavns
```

36.5 Step IV: Registering the WebDAV schema

As mentioned before, registering a new WebDAV namespace is a simple two step process. First, we declare the `IPhoto` schema to be a WebDAV namespace and then we register an adapter for it, making it available for images. In the file `configure.zcml` add the following two directives:

```
1 <configure
2    xmlns="http://namespaces.zope.org/zope"
3    xmlns:dav="http://namespaces.zope.org/dav">
4
5 <dav:provideInterface
```

```
6     for="http://namespaces.zope.org/dav/photo/1.0"
7     interface=".interfaces.IPhoto" />
8
9  <adapter
10    provides=".interfaces.IPhoto"
11    for="zope.app.file.interfaces.IImage"
12    permission="zope.Public"
13    factory=".ImagePhotoNamespace"
14    trusted="True"/>
15
16 </configure>
```

▷ Line 5–7: The `for` attribute specifies the name of the schema as it will be available via WebDAV and `interface` specifies the Zope schema for this namespace.

▷ Line 9–14: Register the adapter from `IImage` to `IPhoto`. Note that the adapter must be trusted, since we are manipulating annotations and need a bare object to be passed as the context of the adapter.

To register the new namespace with the Zope 3 framework, add a file called `photons-configure.zcml` to `package-includes` having the following line:

```
1 <include package="book.photodavns" />
```

You can now restart Zope 3 to make the namespace available.

36.6 Step V: Functional Testing

Now let's see our new namespace in action. Unfortunately, I am not aware of any WebDAV tools that can handle any namespace in a generic fashion. For this reason we will use functional tests for confirming the correct behavior.

In the first step we will test whether `PROPFIND` will (a) understand the namespace and (b) return the right values from the annotation of the image. Here is the complete code for the functional test, which you should place in a file called `ftests.py`.

```
1  import unittest
2  from transaction import get_transaction
3  from xml.dom.minidom import parseString as parseXML
4  from zope.app.file.image import Image
5  from zope.app.dav.ftests.dav import DAVTestCase
6  from book.photodavns.interfaces import IPhoto
7  from book.photodavns import ImagePhotoNamespace
8
9  property_request = '''\
10 <?xml version="1.0" encoding="utf-8" ?>
11 <propfind xmlns="DAV:">
12   <prop xmlns:photo="http://namespaces.zope.org/dav/photo/1.0">
13     <photo:height />
14     <photo:width />
15     <photo:equivalent35mm />
```

```
16      <photo:aperture />
17      <photo:exposureTime />
18    </prop>
19  </propfind>
20  '''
21
22  data = {'height': 768, 'width': 1024, 'equivalent35mm': u'41 mm',
23          'aperture': u'f/2.8', 'exposureTime': 0.031}
24
25  class IPhotoNamespaceTests(DAVTestCase):
26
27      def createImage(self):
28          img = Image()
29          photo = ImagePhotoNamespace(img)
30          for name, value in data.items():
31              setattr(photo, name, value)
32          root = self.getRootFolder()
33          root['img.jpg'] = img
34          get_transaction().commit()
35
36      def test_propfind_fields(self):
37          self.createImage()
38          response = self.publish(
39              '/img.jpg/',
40              env={'REQUEST_METHOD':'PROPFIND',
41                   'HTTP_Content_Type': 'text/xml'},
42              request_body=property_request)
43          self.assertEqual(response.getStatus(), 207)
44          xml = parseXML(response.getBody())
45          node = xml.documentElement.getElementsByTagName('prop')[0]
46
47          for name, value in data.items():
48              attr_node = node.getElementsByTagName(name)[0]
49              self.assertEqual(attr_node.firstChild.data, unicode(value))
50
51  def test_suite():
52      return unittest.TestSuite((
53          unittest.makeSuite(IPhotoNamespaceTests),
54          ))
55
56  if __name__ == '__main__':
57      unittest.main(defaultTest='test_suite')
```

▷ Line 9–20: This is the XML request that will be sent to the Zope 3 WebDAV server. Note that we need to make sure that the first line starts at the beginning of the string, since otherwise the XML parser causes a failure. In the string, we simply request explicitly all attributes of the `photo` namespace.

▷ Line 22–23: Here is the data that is being setup in the annotation and that we expect to receive from the `PROPFIND` request.

▷ Line 27–34: This helper method creates an image and sets the photo data on the image, so that we can access it. Note that we have to commit a transaction at this point, otherwise the image will not be found in the ZODB.

▷ 36–49: First we create the image so that it will be available. Then we just publish our request with a carefully constructed environment. To make the request a `PROPFIND` call, you need to create a environment variable named `REQUEST_METHOD`. Since we send XML as the body of the request, we need to set the content type to "text/xml", which is done with a `HTTP_Content_Type` environment entry.

The answer we receive from the server should be 207, which signalizes that the `PROPFIND` call was successful and the data is enclosed in the body. We then parse the XML body simply using Python's built-in `xml.dom.minidom` package. The rest of the test code simply uses DOM to ensure that all of the requested attributes were returned and the data is correct.

Once you are done with the functional test, you can run it using the usual method:

```
python test.py -vpf --dir src/book/photodavns
```

The `-f` option executes only functional tests. Functional tests are recognized by their module name, which must be `ftests` in comparison to `tests` for regular unit tests.

Exercises

1. Implement `height` and `width` in a way that it uses the `IImage`'s `getImageSize()` method to get the values.

2. JPEG files support EXIF meta-data tags that often already contain the data provided by the `IPhoto` interface, so change the adapter in a way that it first tries to retrieve the data from the image before using annotation data. See `http://topo.math.u-psud.fr/~bousch/exifdump.py` for a Python implementation of a EXIF tag retriever.

CHAPTER 37

USING TALES OUTSIDE OF PAGE TEMPLATES

Difficulty

Newcomer

Skills

- You should be familiar with TALES and know how TALES expressions are used in Zope Page Templates.
- Python is required as usual. Being familiar with the `os` and `os.path` standard module is of advantage, since we use the directory hierarchy as data source in the chapter.

Problem/Task

As you probably know by now, Zope 3 comes with a lot of technologies and many of them can be used outside their common environment. TALES expressions are no exception. One can use TALES in any application without much overhead, as this chapter will demonstrate.

Solution

37.1 Introduction

Throughout the book we used Page Templates and therefore TALES expressions extensively. You might have already thought that using TALES outside templates might be quiet useful, either to provide simple access to objects via path expressions, provide users to enter Python code in a controlled environment or simply to specify logical expressions. The latter use case was frequently applied in Zope 2 applications.

What is there to know about running TALES in an application? TALES mainly consists of three concepts, the expression engine, expressions and contexts.

The expression engine is the object that can compile the expression code and returns an expression object that can be executed/called. It is also responsible for the setup. Here you can register the expressions that should be available in TALES and register objects via names that are always available as base objects. A good example of building up a simple engine is given in the `tales.engine` module.

An expression object is able to handle certain input. The most common one is the path expression (`zope.tales.expressions.PathExpr`), which takes a filesystem or URL-like path and tries to resolve it to a Python object by traversing through the path segments. Another one is the string expression (`zope.tales.expressions.StringExpr`), which simply returns a string, but can contain path expressions, for example "`string:Theobjectidis\$\{context/id\}.`". A final common expression you see a lot is the Python expression (`zope.tales.expressions.PythonExpr`), which is simply able to execute the given expression code as Python and returns the output of the operation. In the "Developing a new TALES expression" chapter I will show you how to create a new TALES expression.

The third component, the context (`zope.tales.tales.Context`), is responsible for providing run-time variables and information for the execution of an expression. When you execute an expression, you always have to provide a context. This object has many other methods, but they are mainly available for TAL and are not required for understanding TALES.

By the way, TALES stands for "Template Attribute Language – Expression Syntax".

37.2 The TALES Filesystem Runner

When I was thinking about a concise example to present TALES expressions, I drew blank for months. Finally I thought about a TALES runner, a program that would simply execute TALES expressions and display the result. But that would be really boring without having any data that can be used for the path expressions and so on.

Then I thought about the filesystem, which would provide a great tree with lots of nodes.

So our first task then is to provide some objects which represent directories and regular files (and ignores other file types). Directories should be simple read-only mapping objects (i.e. behave like dictionaries). Okay that should not be too hard to do. Open a new file called `talesrunner.py` and add the following two classes.

```
1  import os
2
3  class Directory(object):
4
5      def __init__(self, path):
6          self.path = path
7          self.filename = os.path.split(path)[1]
8
9      def __getitem__(self, key):
10         path = os.path.join(self.path, key)
11         if not os.path.exists(path):
12             raise KeyError, "No file '%s' in '%s'" %(key, self.filename)
13         elif os.path.isdir(path):
14             value = Directory(path)
15         else:
16             value = File(path)
17         return value
18
19     def get(self, key, default=None):
20         try:
21             return self.__getitem__(key)
22         except KeyError:
23             return default
24
25     def keys(self):
26         return os.listdir(self.path)
27
28     def items(self):
29         return [(key, self[key]) for key in self.keys()]
30
31     def values(self):
32         return [value for key, value in self.items()]
33
34
35 class File(object):
36
37     def __init__(self, path):
38         self.path = path
39         self.filename = os.path.split(path)[1]
40
41     def read(self):
42         return open(self.path, 'r').read()
```

As you can notice, I did not worry about writing interfaces for this demo. Also, these are really simple implementations and I did not include anything advanced, like creation and modification date, size or permissions.

▷ Line 7: Let's always provide the name of the directory.

▷ Line 9–17: If the requested file does not exist in the directory, raise a `KeyError` saying that the file does not exist. Using the `os.path` module, we can easily determine whether a given key represents a file or directory and create a corresponding object for it accordingly.

▷ 24–25: The keys of a directory are simply all files it contains.

▷ 34–41: There is not much we can say about a file, so we want to at least provide a method that shows its data.

Now that we have a way of providing a nice data tree, we can implement the TALES runner. Simply add the following few lines to the same file:

```
1  import sys
2  from zope.tales.engine import Engine
3  from zope.tales.tales import Context
4
5  if __name__ == '__main__':
6      path = sys.argv[1]
7      context = Context(Engine, {'root': Directory(path)})
8      while 1 == 1:
9          expr = raw_input("TALES Expr: ")
10         if expr == 'exit':
11             break
12         try:
13             bytecode = Engine.compile(expr)
14             print bytecode(context)
15         except Exception, error:
16             print error
```

▷ Line 2–3: For this example the standard engine and context objects are fine. If you want to create your own Engine, because you want to reduce the number of available expression types, you can just look at `zope.tales.engine` to see how the engine was created. It is only a simple method that is easily understandable.

▷ Line 6: When executed, the runner expects a single command line argument, which is the path to the directory that is being used as root.

▷ Line 7: Create a context for the TALES expressions. Make the root directory available under the name `root`.

▷ Line 13–14: One can easily compile any expression by calling `Engine.compile()`. The `bytecode` object is simply an instance of one of the registered expressions. I pointed out before that expressions are executed simply by calling them. The `__call__()` method expects a context, so we pass one in. This will ensure that `root` is available during the execution.

That was easy, wasn't it? Note that you often do not need to mess with `Context`. On the other hand, it is pretty reasonable to expect that people will change the

engine, for example to exclude the python expression, since it is simply too powerful for some applications. Conversely, sometimes you might want to add additional expression types to an engine.

Once you have set `ZOPE3/src` as part of your Python path, you can execute the runner using

```
python talesrunner.py /
```

which uses the Unix root directory as the root for the TALES expressions. Here is an example of a session.

```
$ python talesrunner.py /
TALES Expr: root/keys
['boot', 'dev', 'etc', 'usr', 'lib', 'bin', 'opt', ...]
TALES Expr: exists: root/us
0
TALES Expr: exists: root/usr
1
TALES Expr: root/usr
<__main__.Directory object at 0x4036e12c>
TALES Expr: root/usr/filename
usr
TALES Expr: string: This is the ${root/usr/path} directory.
 This is the /usr directory.
TALES Expr: root/etc/motd/read
Welcome!

TALES Expr: python: root['usr'].keys()
['share', 'bin', 'lib', 'libexec', 'include', ...]
TALES Expr: exit
```

Exercises

1. Include file status information for files and directories. See the `os.stat` module for details on how to get the information.

2. Implement your own engine that does not support `python` and `defer` expressions.

CHAPTER 38

DEVELOPING A NEW TALES EXPRESSION

Difficulty

Contributer

Skills

- Solid knowledge about TAL and TALES as well as page templates is required.
- Detailed insight in Zope 3's RDB integration is of advantage.
- Basic API knowledge of the `ExpressionEngine` and `Context` components is desirable. Optional.

Problem/Task

TAL in combination with TALES provides an incredibly powerful templating system for many types of applications (not only Zope). However, a templating system must be able to adjust to the needs of its various uses. Zope makes extensive use of this flexibility and implements custom versions of the `ExpressionEngine` , `Context` and the expression components.

A way of extending TAL/TALES is to provide additional expressions . Existing expressions include `python`, `string`, and `path` (which is the implicit default). In this chapter we will create an expression that evaluates SQL expressions and returns the result.

Solution

The goal of this chapter is to be able to have TAL code like

```
1 <html tal:define="rdb string:zope.da.PsycopgDA; dsn string:dbi://test">
2   <body>
3     <ul>
4       <li tal:repeat="contact sql: SELECT * FROM contact">
5         <b tal:content="contact/name" />
6       </li>
7     </ul>
8   </body>
9 </html>
```

to be evaluated to

```
1 <html>
2   <body>
3     <ul>
4       <li>[Contact Name 1]</li>
5       <li>[Contact Name 2]</li>
6       ...
7     </ul>
8   </body>
9 </html>
```

▷ Line 1: We tell the system that we want a connection to a PostGreSQL database via the psycopg database adapter, which you can download online. We also tell the system that we would like to connect anonymously to a database called `test`. Alternatively to always specifying the database and the DSN, it will be helpful to be able to specify a Zope Database Adapter object directly:

```
1 <html tal:define="sql_conn string:psycopg_test">
2   ...
3 </html>
```

▷ Line 4: Here we can see that generally an SQL Expression should return a result set containing the various rows bundled as result objects. This is really fortunate, since this is exactly the way Zope database connections return the data anyways.

It should be of course also possible to insert path expressions into the SQL, so the SQL can be dynamic:

```
1 <ul tal:define="name string:Stephan; table string:contact">
2   <li tal:repeat="
3       contact sql: SELECT * FROM ${table} WHERE name = '${name}'">
4     <b tal:content="contact/name" />
5   </li>
6 </ul>
```

Note that expression code should also be responsible for quoting string input correctly.

Next let's have a closer look at the expression component itself. A TALES expression is actually a very simple object, having only a constructor and a call method:

- The constructor takes three arguments, `name`, `expr` and `egine`. The `name` is actually not used and can simply be ignored. The `expr` contains the string that is being evaluated. It contains basically the users "source code". The `engine` is an instance of the `ExpressionEngine` component, which manages all of the different expressions.
- The `__call__()` takes only one argument, namely the `econtext`. The expression context provides expression-external, runtime information, such as declared variables. This allows the expression to behave differently in different contexts and accept custom user input.

One should probably also implement the `__str__` and `__repr__` methods, but they are for cosmetic and debugging purposes only, so they are not that interesting.

38.1 Step I: Implementing the SQL Expression

When I originally wrote the code, I noticed that the SQL expression is almost identical to the string expression, except that instead of returning a string from the `__call__()` method, we evaluate computed string as an SQL statement and return the result. This means that we can safely use the `StringExpr` class as a base for out expression.

While all other code samples of this book are located in the `book` package, this particular package became so popular that it was added to the Zope trunk, though it is not distributed with Zope X3.0. Therefore, create a package called `sqlexpr` in `zope.app`. Inside, create a module called `sqlexpr.py` and add the following code:

```
from zope.component.exceptions import ComponentLookupError
from zope.interface import implements
from zope.tales.interfaces import ITALESExpression
from zope.tales.expressions import StringExpr
from zope.app import zapi
from zope.app.exception.interfaces import UserError
from zope.app.rdb import queryForResults
from zope.app.rdb.interfaces import IZopeDatabaseAdapter, IZopeConnection

class ConnectionError(UserError):
    """This exception is raised when the user did not specify an RDB
    connection."""

class SQLExpr(StringExpr):
    """SQL Expression Handler class"""

    def __call__(self, econtext):
        if econtext.vars.has_key('sql_conn'):
            conn_name = econtext.vars['sql_conn']
            adapter = zapi.queryUtility(IZopeDatabaseAdapter, conn_name)
            if adapter is None:
                raise ConnectionError, \
```

```
23                      ("The RDB DA name, '%s' you specified is not "
24                       "valid." %conn_name)
25          elif econtext.vars.has_key('rdb') and econtext.vars.has_key('dsn'):
26              rdb = econtext.vars['rdb']
27              dsn = econtext.vars['dsn']
28              try:
29                  adapter = zapi.createObject(None, rdb, dsn)
30              except ComponentLookupError:
31                  raise ConnectionError, \
32                        ("The factory id, '%s', you specified in the `rdb` "
33                         "attribute did not match any registered factory." %rdb)
34
35              if not IZopeDatabaseAdapter.providedBy(adapter):
36                  raise ConnectionError, \
37                        ("The factory id, '%s', you specifed did not create a "
38                         "Zope Database Adapter component." %rdb)
39            else:
40              raise ConnectionError, \
41                    'You did not specify a RDB connection.'
42
43          connection = adapter()
44          vvals = []
45          for var in self._vars:
46              v = var(econtext)
47              if isinstance(v, (str, unicode)):
48                  v = sql_quote(v)
49              vvals.append(v)
50          query = self._expr % tuple(vvals)
51          return queryForResults(connection, query)
52
53      def __str__(self):
54          return 'sql expression (%s)' % `self._s`
55
56      def __repr__(self):
57          return '<SQLExpr %s>' % `self._s`
58
59
60  def sql_quote(value):
61      if value.find("\'") >= 0:
62          value = "''".join(value.split("\'"))
63      return "%s" %value
```

▷ Line 5–8: Most TALES expressions do not depend on `zope.app`, which makes them usable outside of Zope. However, for this expression we use much of the Zope relational infrastructure, so that this particular expression depends on `zope.app`.

▷ Line 10–12: Of course it is not guaranteed that the user correctly specified the required variables `rdb`/ `dsn` or `sql_conn`. If an error occurs while retrieving a Zope RDB connection from this data, then this exception is raised. It is a `UserError`, since these exceptions are not due to a system failure but wrong user input.

▷ Line 18–45: It is necessary to figure out, whether we actually have the right variables defined to create/use a database connection.

- Line 18–24: A Zope Database Adapter was specified, so we try to look it up. If no adapter was found, raise a `ConnectionError` exception.
- Line 25–38: If the `rdb`/ `dsn` pair is specified, then we assume the `rdb` is the factory id and we try to initialize the database adapter using the `dsn`. If the `rdb` value is not a valid factory id, we raise a `ConnectionError` (line 32–34). If the created object is not a `IZopeDatabaseAdapter` then also raise a `ConnectionError` (line 36–38).
- Line 39–41: None of the two options was specified, so raise an error.

▷ Line 43: Get a connection from the database adapter by calling it.

▷ Line 44–49: First we evaluate all the path expressions. We then quote all string/unicode values.

▷ Line 50–51: Make final preparations by inserting the path expression results in the query, then we execute the query and return the result.

▷ Line 60–63: The SQL quoting function simply replaces all single quotes with two single quotes, which is the correct escaping for this character.

That's it. The main code for the expression has been written. Now we only need to add the expression to the Zope TAL engine. To do that, create a configuration file and add the following directive:

```
1  <configure
2      xmlns="http://namespaces.zope.org/zope"
3      xmlns:tales="http://namespaces.zope.org/tales"
4      i18n_domain="zope"
5      >
6
7    <tales:expressiontype
8        name="sql"
9        handler=".sqlexpr.SQLExpr"
10       />
11
12 </configure>
```

To insert the new expression to the Zope 3 framework, add a file called `sqlexpr-configure.zcml` to `package-includes` having the following line:

```
1 <include package="zope.app.sqlexpr" />
```

38.2 Step II: Preparing and Implementing the tests

Writing tests for this code is actually quiet painful, since we have to simulate an entire database connection and result objects. Furthermore we have to bring up the Component Architecture, since the RDB connection is created using the `createObject()` function, which uses factories. In `tests.py` add:

```
import unittest

from zope.interface import implements
from zope.component.factory import Factory
from zope.component.interfaces import IFactory
from zope.component.tests.placelesssetup import PlacelessSetup
from zope.tales.tests.test_expressions import Data
from zope.tales.engine import Engine

from zope.app.tests import ztapi
from zope.app.rdb.interfaces import IZopeDatabaseAdapter, IZopeConnection
from zope.app.rdb.tests.stubs import ConnectionStub
from zope.app.sqlexpr.sqlexpr import SQLExpr, ConnectionError


class AdapterStub(object):
    implements(IZopeDatabaseAdapter)

    def __init__(self, dsn):
        return

    def __call__(self):
        return ConnectionStub()

class ConnectionStub(object):
    implements(IZopeConnection)

    def __init__(self):
        self._called = {}

    def cursor(self):
        return CursorStub()

class CursorStub(object):

    description = (('id', 0, 0, 0, 0, 0, 0),
                   ('name', 0, 0, 0, 0, 0, 0),
                   ('email', 0, 0, 0, 0, 0, 0))


    def fetchall(self, *args, **kw):
        return ((1, 'Stephan', 'srichter'),
                (2, 'Foo Bar', 'foobar'))

    def execute(self, operation, *args, **kw):
        if operation != 'SELECT num FROM hitchhike':
            raise AssertionError(operation, 'SELECT num FROM hitchhike')


class SQLExprTest(PlacelessSetup, unittest.TestCase):

    def setUp(self):
        super(SQLExprTest, self).setUp()
        ztapi.provideUtility(IFactory, Factory(AdapterStub),
                             'zope.da.Stub')
        ztapi.provideUtility(IFactory, Factory(lambda x: None),
                             'zope.Fake')
        ztapi.provideUtility(IZopeDatabaseAdapter, AdapterStub(''),
```

```
59                              'test')
60
61      def test_exprUsingRDBAndDSN(self):
62          context = Data(vars = {'rdb': 'zope.da.Stub', 'dsn': 'dbi://test'})
63          expr = SQLExpr('name', 'SELECT num FROM hitchhike', Engine)
64          result = expr(context)
65          self.assertEqual(1, result[0].id)
66          self.assertEqual('Stephan', result[0].name)
67          self.assertEqual('srichter', result[0].email)
68          self.assertEqual('Foo Bar', result[1].name)
69
70      def test_exprUsingSQLConn(self):
71          context = Data(vars = {'sql_conn': 'test'})
72          expr = SQLExpr('name', 'SELECT num FROM hitchhike', Engine)
73          result = expr(context)
74          self.assertEqual(1, result[0].id)
75          self.assertEqual('Stephan', result[0].name)
76          self.assertEqual('srichter', result[0].email)
77          self.assertEqual('Foo Bar', result[1].name)
78
79      def test_exprUsingRDBAndDSN_InvalidFactoryId(self):
80          context = Data(vars = {'rdb': 'zope.da.Stub1', 'dsn': 'dbi://test'})
81          expr = SQLExpr('name', 'SELECT num FROM hitchhike', Engine)
82          self.assertRaises(ConnectionError, expr, context)
83
84      def test_exprUsingRDBAndDSN_WrongFactory(self):
85          context = Data(vars = {'rdb': 'zope.Fake', 'dsn': 'dbi://test'})
86          expr = SQLExpr('name', 'SELECT num FROM hitchhike', Engine)
87          self.assertRaises(ConnectionError, expr, context)
88
89      def test_exprUsingSQLConn_WrongId(self):
90          context = Data(vars = {'sql_conn': 'test1'})
91          expr = SQLExpr('name', 'SELECT num FROM hitchhike', Engine)
92          self.assertRaises(ConnectionError, expr, context)
93
94      def test_noRDBSpecs(self):
95          expr = SQLExpr('name', 'SELECT num FROM hitchhike', Engine)
96          self.assertRaises(ConnectionError, expr, Data(vars={}))
97
98
99  def test_suite():
100     return unittest.TestSuite((
101         unittest.makeSuite(SQLExprTest),
102         ))
103
104 if __name__ == '__main__':
105     unittest.main(defaultTest='test_suite')
```

▷ Line 16–23: Implement a database adapter stub that only can create connection stubs. We even ignore the DSN.

▷ Line 25–32: This connection object does not implement the entire interface of course; we only need the `cursor()` method here.

▷ Line 34–47: Whatever SQL query will be made, only a simple result is returned having two rows with three entries, `id`, `name` and `email`. If a query

is successful then we should expect this result. Also note that only the query `SELECTnumFROMhitchhike` is considered a valid SQL statement.

- ▷ Line 54–57: Create two factories. The first one is a factory for a valid Zope Database Adapter component, so other one is a dummy factory. Having the dummy factory will allow us to cause one of the anticipated failures.
- ▷ Line 58–59: Register an existing Database Adapter instance, so that the use of `sql_conn` can be tested.
- ▷ Line 61–68: This is a simple test using `rdb` and `dsn` to see whether a simple query executes correctly.
- ▷ Line 70–77: Another simple test, this time using the `sql_conn` variable.
- ▷ Line 79–96: These tests all attempt to cause `ConnectionError` exceptions. All possible cases are covered.

Now you should run the tests and make sure they work and fix errors if necessary.

38.3 Step III: Trying our new expression in Zope

The following walk-through assumes that you have installed and started PostGreSQL. Furthermore, you should have installed the `psycopgda` database adapter in your Python path somewhere (i.e. `ZOPE3/src`. Before we can come back to Zope, you will need to create a user `zope3` and a database called `test` using the commands from the shell:

```
createuser zope3
createdb test
```

The user should be the same as the one running Zope 3 and the database should be created by the user `zope3`. Now enter the database using `psqltest` and add a table called `contact` having at *least* one column called `name`. Add a couple entries to the table.

```
1 CREATE TABLE contact (name varchar);
2 INSERT INTO contact VALUES ('Stephan');
3 INSERT INTO contact VALUES ('Claudia');
4 INSERT INTO contact VALUES ('Jim');
```

Next restart Zope. Go to the management interface, add a `ZPTPage` and add the following content:

```
1 <html tal:define="rdb string:zope.da.PsycopgDA; dsn string:dbi://test">
2   <body>
3     <ul>
4       <li tal:repeat="contact sql: SELECT * FROM contact">
```

```
5           <b tal:content="contact/name" />
6         </li>
7       </ul>
8     </body>
9   </html>
```

Once you saved the code, you can click on the `Preview` tab and you should see a bulleted list of all your `contact` entries names. Feel free to test the path expression functionality and the usage of a database connection directly as well.

Exercises

1. Implement an expression that evaluates regular expressions. There are two interesting functions that one can do with regular expressions, namely find/match and replace (manipulation). For this exercise just implement find, which returns a list of matches. The syntax might look as follows:

```
1 <html tal:define="foo regex: path/to/old/foo, '([a-zA-Z]*) ?'" />
```

CHAPTER 39

SPACESUITS – OBJECTS IN HOSTILE ENVIRONEMENTS

Difficulty

Core Developer

Skills

- Be familiar with the general usage of the Zope 3 security mechanism.
- Know how checkers and proxies work. Optional.

Problem/Task

Zope 3 can contain serious data or provide access to data that needs to be protected. This requirement is even amplified, if you think about the fact that actually everyone can access Zope from any Web browser. One of Zope 3's strengths is a solid security system, which provides all the flexibility needed to implement any complex security requirements.

Solution

39.1 Getting started ...

From a security point of view, Zope distinguishes between trusted and untrusted environments and code. Everything that the programmer has complete control over,

like filesystem-based Python modules, are considered trusted. Data or objects that use input from the user, such as URLs or HTML form data, is always considered untrusted and one should not blindly trust this data.

In order to be able to let any object float around in an untrusted universe, they receive a spacesuit to protect the object. The spacesuit, known as a security proxy, is a transparent object that encapsulates an object. The security proxy controls all attribute accesses and mutations. Before access to an attribute is granted, the proxy checks with the interaction (which is the realization of the security policy), if the requesting party (the participations, which contains the principal/user) making the request has permission to do so. But hold on, where did this interaction suddenly come from? The interaction is stored as a thread-global variable via the security management (`zope.security.management`) interface. The interaction can be compared to the physical laws of the universe, which tells you if an action between an actor (participation) and an object is possible.

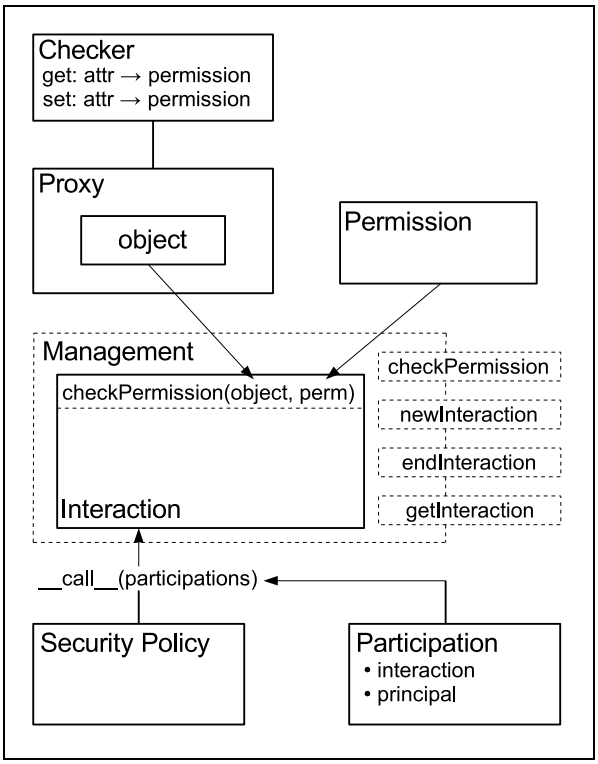


Figure 39.1: This is a diagram containing all major security-relavant components. The lines between them try to show how they are all connected.

One component that we only mentioned briefly but is very important is the security policy, which provides a blue print for the laws of the universe. But only when we add some actors (participations) to these laws, the potential is realized. Therefore, the security policy has only one methods, `__call__(*participations)`, which takes a list of participations and returns an interaction (a realization of itself). Since security policies are pluggable in Zope, the default security policy is shipped in a separate pacakge called `zope.app.securitypolicy`. Zope's policy also provides the concept of roles to Zope (roles group permissions together) and handles the complex structures that keep track of the various security declarations of the system. For more details on the policy, see the interfaces and code in `zope.app.securitypolicy`.

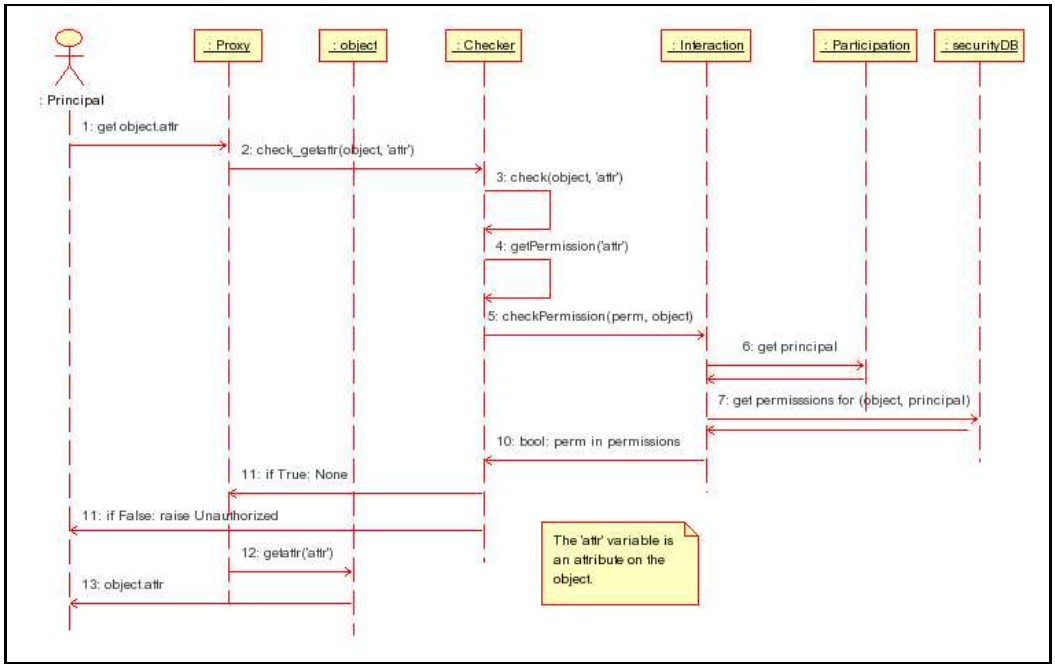


Figure 39.2: Here you see the calls that are done in the security machinery to get the value of a simple attribute.

The goal of the chapter is to develop a very simplistic application, specifically an adventure game, that first runs completely without security. Then, almost without any intrusion, a security framework is built around it, implementing a custom security policy and ovbiously going through all the other steps to get security running.

39.2 The Labyrinth Game

The little game we will produce has a small maze of rooms through which a player can walk.

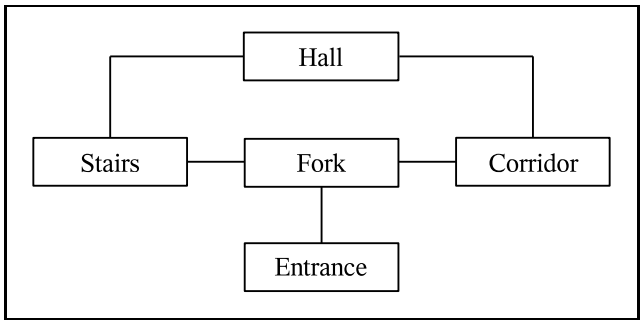


Figure 39.3: The Labyrinth Maze

The player keeps track of the room he is in. The room can have connections to other rooms in the four points of compass. Therefore a short implementation of the requirements above could look like the classes below.

```
1  class Person(object):
2
3      def __init__(self, id):
4          self.id = id
5          self.room = None
6
7      def goTo(self, direction):
8          assert direction in ('north', 'south', 'east', 'west'), \
9                 '"%s" is not a valid direction' %direction
10         room = getattr(self.room, direction, None)
11         if room is None:
12             print 'There is no room %s of here.' %direction
13         else:
14             print room.description
15             self.room = room
16
17 class Room(object):
18
19     def __init__(self, id, description):
20         self.id = id
21         self.description = description
22         self.north = self.south = self.east = self.west = None
```

▷ Line 7–15: A convenience method that will move the user to the next room.

Note that I did not worry about writing interfaces for the objects, since they are trivial and the interfaces would not be useful for anything.

Next we have to develop a function that will do all of the dirty setup work for us, like create actual rooms, connect them and finally create a player with an assigned starting room.

```
def setupWorld():
    # Create the rooms
    entrance = Room('entrance', 'The entrance of the labyrinth')
    fork = Room('fork', 'The big decision. Do I go east or west.')
    stairs = Room('stairs', 'Some long dark stairs.')
    hall = Room('hall', 'A cathedral-like hall.')
    corridor = Room('corridor', 'A long corridor')

    # Connect the rooms
    entrance.north = fork
    fork.south, fork.west, fork.east = entrance, stairs, corridor
    stairs.east, stairs.north = fork, hall
    corridor.west, corridor.north = fork, hall
    hall.west, hall.east = stairs, corridor

    # Setup player
    player = Person('player')
    player.room = entrance
    return player
```

The final step is to develop the game loop that allows the player to move. The game will support six fundamental commands: exit, info, north, south, east, and west. When an error occurs, the game should not crash but try to display the error as gracefully as possible.

```
def main():
    player = setupWorld()
    command = ''
    while command != 'exit':
        try:
            if command == 'info':
                print player.room.description
            elif command:
                player.goTo(command)

        except Exception, e:
            print '%s: %s' %(e.__class__.__name__, e)

        command = raw_input('Command: ')

if __name__ == '__main__':
    main()
```

Now, if you save all of the code in a module file named `labyrinth.py`, then you can just run the game with

```
python labyrinth.py
```

Here is the output of a simple session:

```
# python labyrinth.py
Command: info
The entrance of the labyrinth
```

```
Command: north
The big decision. Do I go east or west.
Command: west
Some long dark stairs.
Command: north
A cathedral-like hall.
Command: east
A long corridor
Command: west
The big decision. Do I go east or west.
Command: south
The entrance of the labyrinth
Command: exit
#
```

39.3 Securing the Labyrinth

Now that we have a running game, let's secure it. But what does "secure" mean in the context of the game? It means that the player needs to have the permission to enter the room. And we will not give her/him access to the `corridor`. We will also limit the security to simply have an allow or deny policy. The following code is developed in a new file called `labyrinth_security.py`.

The first step is to declare the permissions that will be available to the security for use. In our case this is just the `Allow` permission, since denying is just not having the former permission. So all we need is

```
Allow = 'allow'
```

Next we have to build a permission database that stores the information on which rooms the person/player is allowed to enter. This can be done with a simple dictionary that maps a `roomid` to the principals that are allowed to access that room.

```
permissions = {}

def allowPerson(roomid, personid):
    """Allow a particular person in a room."""
    perms = permissions.setdefault(roomid, [])
    perms.append(personid)
```

We have to implement the security-related components now. Let's start with the participation, which can simply fulfill its interface with the following trivial implementation.

```
from zope.security.interfaces import IParticipation

class PersonParticipation(object):

    implements(IParticipation)

    def __init__(self, person):
        self.principal = person
        self.interaction = None
```

Finally we need to implement the security and the interaction. Since the interaction is just a realization of the security policy, the interaction is simply an instance of the security policy class. Most of the security policy class is already implemented by the simple policies, so we can just reuse it and concentrate on only implementing `checkPermission()`.

```
1  from zope.security import simplepolicies
2
3  class SecurityPolicy(simplepolicies.ParanoidSecurityPolicy):
4      """The Labyrinth's access security policy."""
5
6      def checkPermission(self, permission, object):
7          """See zope.security.interfaces.ISecurityPolicy"""
8          assert permission is Allow
9          allowed = permissions.get(object.id, [])
10         for participation in self.participations:
11             if not participation.principal.id in allowed:
12                 return False
13         return True
```

▷ Line 6: This security policy can only handle the `Allow` permission, so we assert this right away.

▷ Line 7: Get a list of principals that are allowed in this room.

▷ Line 8–11: Every participant in the interaction must have the permission for the action. For example, all construction workers that are required to renovate a house need a key to the house to do the renovation.

Next we need to setup the security, which we do in a function called `setupSecurity(player)`.

```
1  import labyrinth
2  from zope.security import checker, management
3
4  def setupSecurity(player):
5      # Setup security
6      management.setSecurityPolicy(SecurityPolicy)
7      room_checker = checker.NamesChecker(
8          ('description', 'north', 'south', 'west', 'east'), Allow)
9      checker.defineChecker(labyrinth.Room, room_checker)
10
11     # Allow the player everywhere but the corridor
12     allowPerson('entrance', player.id)
13     allowPerson('fork', player.id)
14     allowPerson('stairs', player.id)
15     allowPerson('hall', player.id)
16
17     # Add the player as a security manager and provide the player with a
18     # secure room
19     management.newInteraction(PersonParticipation(player))
20     proxied_room = checker.selectChecker(player.room).proxy(player.room)
21     player.room = proxied_room
22     return player
```

▷ Line 5–6: Register our security policy.

▷ Line 7–9: Create a `Checker` for the room, allowing all available attributes of the `Room` class. In the second step we associate the checker with the `Room`. It is now up to the player, whether s/he has the Allow permission for a room. You could compare this to local permissions in Zope.

▷ Line 11–15: Allow the player to go anywhere but the `corridor`.

▷ Line 19: Create and store a new interaction using a player-based participation. Note that the security policy must be set before the interaction is created.

▷ Line 20–21: Put a proxy around the entrance. Note that any object that is returned by any method or attribute of the entrance is also proxied, so that from now on all rooms the player interacts with are proxied.

The final step is to hook up the security and run the game.

```
1 if __name__ == '__main__':
2     oldSetupWorld = labyrinth.setupWorld
3     labyrinth.setupWorld = lambda : setupSecurity(oldSetupWorld())
4     labyrinth.main()
```

▷ Line 2–3: First we make a copy of the old setup. Then we create a new `setupWorld()` function that adds the setup of the security. This monkey patch is the only intrusion to the original code that we wrote in the previous section.

▷ Line 4: Run the game.

Finally, we are done! Before you can run the secure game, you have to set the python path correctly to `ZOPE3/src`. Here is a sample transcript of playing the game.

```
# python labyrinth_security.py
Command: north
The big decision. Do I go east or west.
Command: east
Unauthorized: You are not authorized
Command: west
Some long dark stairs.
Command: north
A cathedral-like hall.
Command: east
Unauthorized: You are not authorized
Command: exit
#
```

Note how we get an `Unauthorized` error, if we try to access the corridor.

This completes my introduction of the Zope security mechanism. A more complex example can be found in `zope.security.example`.

CHAPTER 40

THE LIFE OF A REQUEST

Difficulty

Contributor

Skills

- You should be well–familiar with the Zope 3 framework.
- Having a general idea of Internet server design is also very helpful.

Problem/Task

When developing Zope 3 applications, the coder is commonly dealing with the `request` object to create views without thinking much about the details on how the request gets into the view and what happens with the response that is constructed in it. And this is fine, since it is often not necessary to know. But sometimes one needs to write a custom server or change the behavior of the publisher. In these cases it is good to know the general design of the Zope servers and publishers. This chapter takes you on the journey through the life of a request using the browser (special HTTP) request as an example.

Solution

40.1 What is a Request

The term "request" appears often when talking about Zope 3. But what is a request? In technically concrete situations we usually refer to objects that implement

IRequest. These objects are responsible to embed protocol-specific details and represent the protocol semantics to provide them for usage by presentation components.

It is not enough to only think of request objects, though. For me, anything that the client sends to the server after connection negotiations is considered a request. So on the very lowest level, when the user agent (in this case the classic Web browser) sends the HTTP string

```
1 GET /index.html HTTP/1.1
```

it could be considered a request (raw) as well.

40.2 Finding the Origin of the Request

Now that we have an idea what the request is, let's take a more technical approach and find out how connections are handled technically and how a request is born in the midst of much redirection and many confusing abstraction layers.

When a server starts up, it binds a socket to an address on the local machine (bind(address)) and then starts to listen for connections by calling listen(backlog). When an incoming connection is detected, the accept() method is called, which returns a connection object and the address of the computer to which the connection was made. All of this is part of the standard Python socket library and is documented in the zope.server.interfaces.ISocket interface.

The server, which is mainly an IDispatcher implementation, has a simple interface to handle the specific events, by calling its corresponding handle_⟨event⟩() methods. A complete list of all events that are managed this way is given in the IDisplatcherEventHandler interface. So when a connection comes in, handle_accept() is called, which is overridden in the zope.server.serverbase.ServerBase class around line 130. This method tries to get the connection by calling accept() (see previous paragraph). If the connection was successfully created, it is used to create a ServerChannel, which is the next level of abstraction. Most of the other dispatcher functionality is provided by the standard async.dispatcher, which fully implements IDispatcher.

At this stage the channel is taking over, which is just another dispatcher. So the channel starts to collect the incoming data (see received(data)) and sends it right away to the request parser, which is an instance of a class specified as parser_class. Obviously, this class will be different for every server implementation. The way the parser functions is not that important. All we have to know is that it implements IStreamConsumer, which has a way of saying when it has completed parsing a request. Once all of the data is received, the IServerChannel method receivedCompleteRequest(req) is called, whose goal is to schedule the request to be executed. But only one request of the channel can be running at a time. So if

the channel is busy, then we need to queue the request, until the running task is completed.

Whenever the channel becomes available (see `end_task(close)`, the next request from the queue is taken and converted to an `ITask` object. The task is then immediately sent to the server for execution using `IServer.addTask(task)`. There it is added to a task dispatcher (see `ITaskDispatcher`), which schedules the task for execution. But why all this redirection through a task and a task dispatcher? Until now, all the code ran on a single thread. But in order to scale the servers better and to support long-running requests without blocking the entire server, it is necessary to be able to start several threads to handle the requests. It is now up to the `ITaskDispatcher` implementation to decide how to spread the requests. Theoretically, it could even consult other computers to execute a task. By default, we use the `zope.server.taskthreads.ThreadedTaskDispatcher` though. Using its `setThreadCount(count)` method, the Zope startup code is able specify the maximum amount of threads running at a time.

Once, it is the task's turn to be serviced, the task dispatcher calls `ITask.service()` which should finally execute the request. Specifically, when the `HTTPTask` is serviced, the method `executeRequest(task)` of the `HTTPServer` is called. The `zope.server.http.publisherhttpserver.PublisherHTTPServer`, which is the one used for Zope 3, creates a `IHTTPRequest` object from the task and publishes the request with `zope.publisher.publish(request)`. The server has an attribute `request_factory` in which the request class that is used to create the request is stored.

So what did the system accomplish so far? We have taken an incoming connection, read all the incoming data and parsed it, scheduled it for execution, and finally created a request that was published with the Zope 3 publisher. Except for the last step, there was nothing Zope-specific about this code, so that all of this could be replaced by any other Web server, like Twisted's.

40.3 The Request and the Publisher

With the birth of the request and its start to walk the path of a request's life by entering the publisher using `zope.publisher.publish.publish()`, it also enters a Zope-pure domain. In fact, Zope does not really care how a request was created, as long as it implements `IRequest`. For example, when functional tests are executed, they create the request purely from fictional data and pass it through the publisher to analyze the response afterwards.

From a high-level point of view, the publisher's `publish()` method is responsible of interpreting the request's information and cause the correct actions. It starts

out by traversing the given object path to an actual object, then call the object and finally writing the result in the response. Everything else in this method is about handling errors and exceptions as well as providing enough hooks for other components to step in.

The `publish()` method is so central to the entire Zope 3 framework, that we will now go through it very carefully trying to understand each step's purpose. Wherever necessary, we will take a rest and examine side paths closer. It might be of advantage to open the `zope.publisher.publish` module at this point, so that it is easier to follow the text.

The work of the `publish()` method starts with a infinite while–loop. The first step inside the loop is to get the publication.

The publication provides the publisher with hooks to accomplish application-specific tasks, related to data storages, transactions and security. The default implementation is `DefaultPublication`, which is located in `zope.publisher.base` and can be used by software that do not make use of the entire Zope framework. For Zope 3, however, there is a specific Zope implementation in `zope.app.publication.zopepublication`.

Now, wrapped inside three `try/except` statements, we tell the request to look at its data and process whatever needs to be processed. In the case of a browser request, like the one we use as example, the `processInputs()` tries to parse all HTML form data that might have been sent by the browser and convert it to Python objects.

The next step is to convert the request's object path to an object, a process known as traversal. Besides calling all the event–hooks, the first step of the traversal process is to determine the application or, in other words, the object root. For a common Zope 3 installation, the application is of course the root of the ZODB. Then we use the request's `traverse(object)` method to get to the desired object. Let's have a closer look at this method for the `BrowserRequest`.

First of all we notice that the `BrowserRequest`'s traverse method does not do any of the heavy lifting, but only covers a few browser-specific corner cases, like picking a default view and using the HTML form data (by inspecting form variable names for the suffix ":method") for possible additional traversal steps. It turns out that the `BaseRequest`'s traverse method does all the work. At the beginning of the method there are several private attributes that are being pulled into the local namespace and setup.

- `publication`: It is simply the `publication` object from before, which gives us access to the application-specific functionality.

- `_traversal_stack`: A simple stack (i.e. list) of names that must be traversed. These names came from the parsed path of the URL. For example "/path/-

to/foo/bar/index.html" would be parsed to `['path','to','foo','bar','index.html]`.

- `_traversed_names`: This is a list of names that have been already successfully traversed. The names are simply the entries coming from `_traversal_stack`.
- `_last_obj_traversed`: This variable keeps track of the last object that was found in the traversal process.

Now we just work through the traversal stack until it has been completely emptied. The interesting call here is the `publication.traverseName(request,object,name)` which tries to determine the next object using the name from the traversal stack and the request. The `traverseName()` method can be very complex. The Zope 3 application version, found in `zope.app.publication.publicationtraverse.PublicationTraverse`, must be able to handle namespaces ("++namespace++"), views ("@@") and pluggable traverser lookups, so that objects can implement their own traversers. To discuss the details of this method would be beyond the goal of this chapter.

If everything goes well, and no exception was raised, meaning that the object specified in the path was found, the `traverse()` method returns the found object and we are back in the publisher's `publish()` function. The next step is to execute the object.

Calling the object assumes that the object is callable in the first place. Therefore, the traversal process should always end in a view or a method on a view. But since all common content objects have browser-specific default views, we are guaranteed that the object is callable. For other presentation types, similar default options exist. Even though the object is formally executed by calling `publication.callObject(request,object)`, eventually `mapply()` is called, which is defined in the `zope.publisher.publish` module. `mapply()` does not just call the object, but takes great care of determining the argument and finding values for them.

When an object is called, it can either write the result directly to the request's response object or return a result object. In the latter case, the `publish()` method adds the result to the body of the response. Here it is assumed that the result object is a unicode string. For the Zope application the `afterCall(request)` execution is of importance, since it commits the transaction to the ZODB. This process can cause a failure, so it is very important that we do not return any data to the server until the transaction is committed.

When all this has successfully finished, we call `outputBody()` on the response, which sends the data out to the world going through the task, channel and eventually through the socket to the connected machine. Note that the `output(data)` method,

which is called from `outputBody`, is responsible for converting the internally kept unicode strings to valid ASCII using an encoding. If no encoding was specified, "UTF-8" is used by default.

Once the response has sent out its data, the request is closed by calling `close()` on itself, which releases all locks on resources. This will also finish the running task, close the channel and eventually disconnect the socket. This marks the end of the request.

Let's now look at some of the possible failure scenarios. The most common failure is a ZODB write conflict, in which case we simply want to rollback the transaction and retry the request again. But where does the `Retry` error come from, when the ZODB raises a `ConflictError`? A quick look in the publication's `handleException()` method reveals, that if a write conflict error is detected, it is logged and afterwards a `Retry` exception is raised, so that the next exception handler is used. Here we simply reset the request and the response and allow the publishing process to start all over again (remember, we have an everlasting while loop over all of this code).

In general, though, exceptions are handled by the `handleException()` method, which logs the error and even allows them to be formatted in the appropriate output format using a view. See the chapter on "Changing Standard Exception Views" for details on how to define your own views on exceptions.

This concludes our journey through the life of a request. Sometimes I intentionally ignored details to stay focused and not confuse you. If you are interested, you will find that the interfaces of the various involved components serve well as further documentation, especially for the publisher.

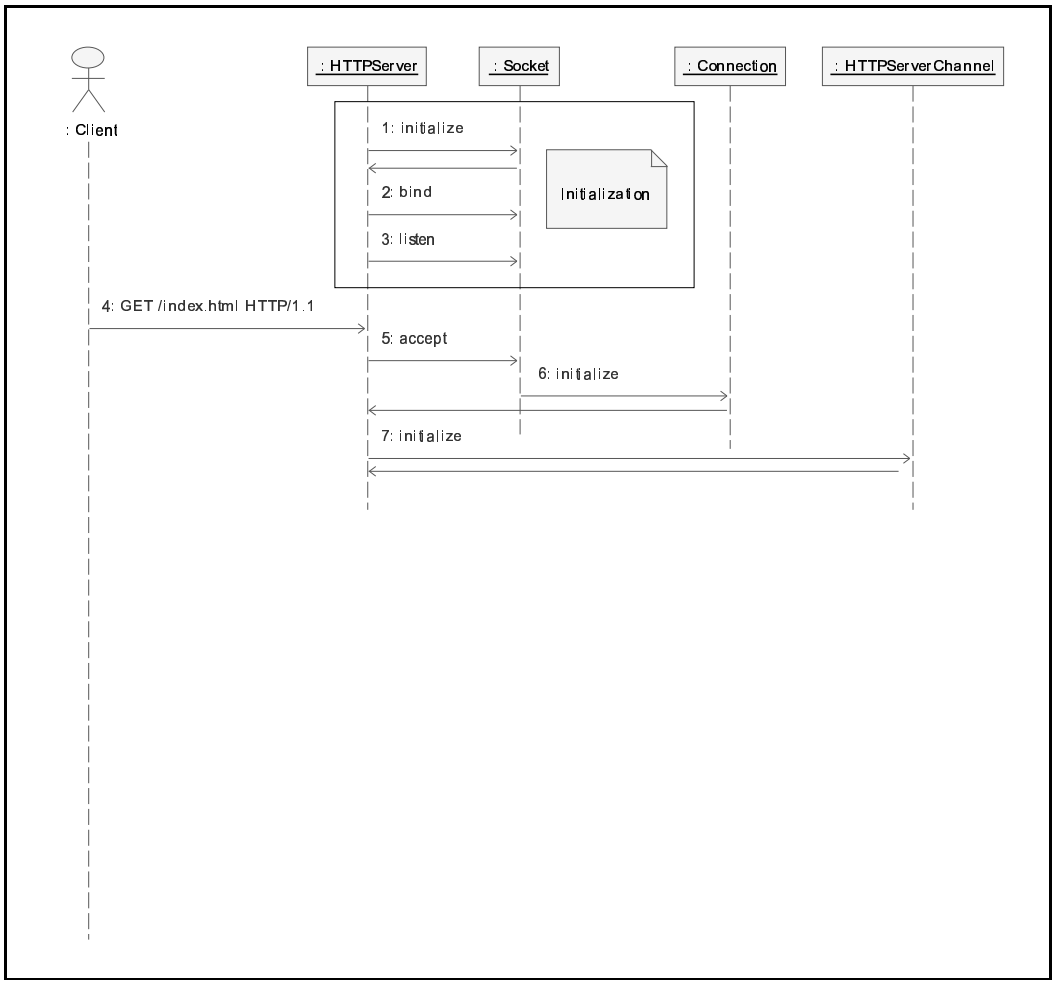


Figure 40.1: Low-Level setup of a connection.

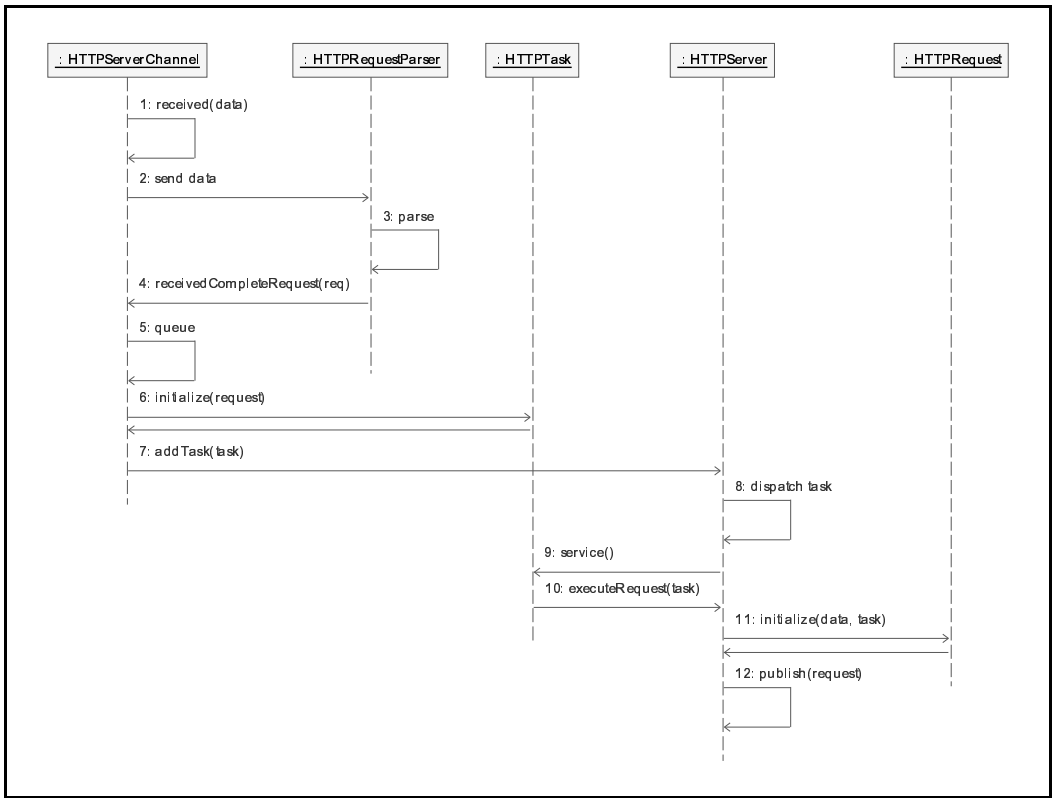


Figure 40.2: From data reception to the initialization of the request object

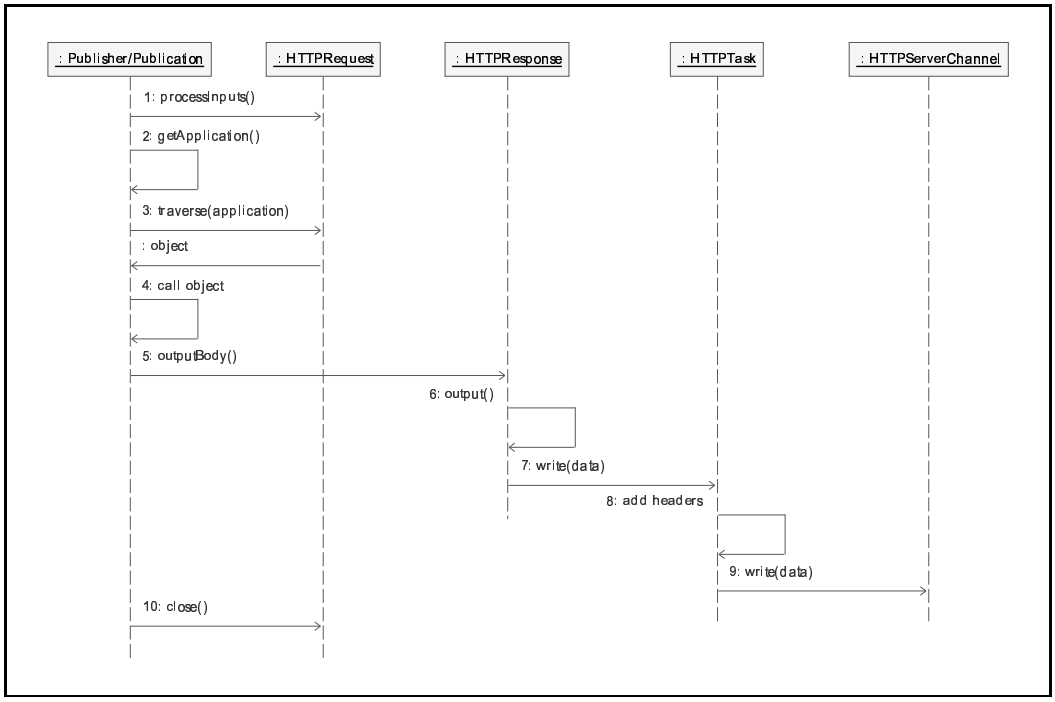


Figure 40.3: The Request in the Publisher.

PART VII

Writing Tests

Writing tests for every bit of functionality is of utmost importance in Zope 3. Testing can be done in many ways, from Java-like unit tests to pythonic doc- testing.

Chapter 41: Writing Basic Unit Tests

This chapter shows you how to develop basic unit tests for your code and explains in detail how various pieces of functionality can be tested.

Chapter 42: Doctests: Example-driven Unit Tests

Sometimes regular unit tests are not as instructive to someone reviewing the code. It is shown how example-driven doctests can become useful in these cases.

Chapter 43: Writing Functional Tests

Unit tests are great for testing isolated components, but are impractical for testing entire sections of the system. For these type of tests we use a functional testing framework, which is introduced here in some detail.

Chapter 44: Creating Functional Doctests

For the same reason doctests were developed to supercede unit tests, functional doctests are intended to be a more descriptive solution to develop functional tests.

Chapter 45: Writing Tests against Interfaces

If an interface is commonly implemented multiple times, it is a good idea to write tests directly against the interface as a base for the implementation tests. This chapter will show you how to do that and expand on the motivation.

CHAPTER 41

WRITING BASIC UNIT TESTS

Difficulty

Newcomer

Skills

- All you need to know is some Python.

Problem/Task

As you know by now, Zope 3 gains its incredible stability from testing any code in great detail. The currently most common method is to write unit tests. This chapter introduces unit tests – which are Zope 3 independent – and introduces some of the subtleties.

Solution

41.1 Implementing the Sample Class

Before we can write tests, we have to write some code that we can test. Here, we will implement a simple class called `Sample` with a public attribute `title` and `description` that is accessed via `getDescription()` and mutated using `setDescription()`. Further, the description must be either a regular or unicode string.

Since this code will not depend on Zope, open a file named `test_sample.py` anywhere and add the following class:

```
1  Sample(object):
2      """A trivial Sample object."""
3
4      title = None
5
6      def __init__(self):
7          """Initialize object."""
8          self._description = ''
9
10     def setDescription(self, value):
11         """Change the value of the description."""
12         assert isinstance(value, (str, unicode))
13         self._description = value
14
15     def getDescription(self):
16         """Change the value of the description."""
17         return self._description
```

▷ Line 4: The `title` is just publicly declared and a value of `None` is given. Therefore this is just a regular attribute.

▷ Line 8: The actual description string will be stored in `_description`.

▷ Line 12: Make sure that the description is only a regular or unicode string, like it was stated in the requirements.

If you wish you can now manually test the class with the interactive Python shell. Just start Python by entering `python` in your shell prompt. Note that you should be in the directory in which `test_sample.py` is located when starting Python (an alternative is of course to specify the directory in your `PYTHONPATH`.)

```
1  >>> from test_sample import Sample
2  >>> sample = Sample()
3  >>> print sample.title
4  None
5  >>> sample.title = 'Title'
6  >>> print sample.title
7  Title
8  >>> print sample.getDescription()
9
10 >>> sample.setDescription('Hello World')
11 >>> print sample.getDescription()
12 Hello World
13 >>> sample.setDescription(None)
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in ?
16   File "test_sample.py", line 31, in setDescription
17     assert isinstance(value, (str, unicode))
18 AssertionError
```

As you can see in the last test, non-string object types are not allowed as descriptions and an `AssertionError` is raised.

41.2 Writing the Unit Tests

The goal of writing the unit tests is to convert this informal, manual, and interactive testing session into a formal test class. Python provides already a module called `unittest` for this purpose, which is a port of the Java-based unit testing product, JUnit, by Kent Beck and Erich Gamma. There are three levels to the testing framework (this list deviates a bit from the original definitions as found in the Python library documentation. [1]).

The smallest unit is obviously the "test", which is a single method in a `TestCase` class that tests the behavior of a small piece of code or a particular aspect of an implementation. The "test case" is then a collection tests that share the same setup/inputs. On top of all of this sits the "test suite" which is a collection of test cases and/or other test suites. Test suites combine tests that should be executed together. With the correct setup (as shown in the example below), you can then execute test suites. For large projects like Zope 3, it is useful to know that there is also the concept of a test runner, which manages the test run of all or a set of tests. The runner provides useful feedback to the application, so that various user interaces can be developed on top of it.

But enough about the theory. In the following example, which you can simply put into the same file as your code above, you will see a test in common Zope 3 style.

```
1  import unittest
2
3  class SampleTest(unittest.TestCase):
4      """Test the Sample class"""
5
6      def test_title(self):
7          sample = Sample()
8          self.assertEqual(sample.title, None)
9          sample.title = 'Sample Title'
10         self.assertEqual(sample.title, 'Sample Title')
11
12     def test_getDescription(self):
13         sample = Sample()
14         self.assertEqual(sample.getDescription(), '')
15         sample._description = "Description"
16         self.assertEqual(sample.getDescription(), 'Description')
17
18     def test_setDescription(self):
19         sample = Sample()
20         self.assertEqual(sample._description, '')
21         sample.setDescription('Description')
22         self.assertEqual(sample._description, 'Description')
23         sample.setDescription(u'Description2')
24         self.assertEqual(sample._description, u'Description2')
25         self.assertRaises(AssertionError, sample.setDescription, None)
26
27
```

[1] http://www.python.org/doc/current/lib/module-unittest.html

```
28 def test_suite():
29     return unittest.TestSuite((
30         unittest.makeSuite(SampleTest),
31         ))
32
33 if __name__ == '__main__':
34     unittest.main(defaultTest='test_suite')
```

▷ Line 3–4: We usually develop test classes which must inherit from `TestCase`. While often not done, it is a good idea to give the class a meaningful docstring that describes the purpose of the tests it includes.

▷ Line 6, 12 & 18: When a test case is run, a method called `runTests()` is executed. While it is possible to overrride this method to run tests differently, the default option will look for any method whose name starts with `test` and execute it as a single test. This way we can create a "test method" for each aspect, method, function or property of the code to be tested. This default is very sensible and is used everywhere in Zope 3.

Note that there is no docstring for test methods. This is intentional. If a docstring is specified, it is used instead of the method name to identify the test. When specifying a docstring, we have noticed that it is very difficult to identify the test later; therefore the method name is a much better choice.

▷ Line 8, 10, 14, ...: The `TestCase` class implements a handful of methods that aid you with the testing. Here are some of the most frequently used ones. For a complete list see the standard Python documentation referenced above.

- `assertEqual(first,second[,msg])`
  Checks whether the `first` and `second` value are equal. If the test fails, the `msg` or `None` is returned.
- `assertNotEqual(first,second[,msg])`
  This is simply the opposite to `assertEqual()` by checking for non-equality.
- `assertRaises(exception,callable,...)`
  You expect the `callable` to raise `exception` when executed. After the `callable` you can specify any amount of positional and keyword arguments for the `callable`. If you expect a group of exceptions from the execution, you can make `exception` a tuple of possible exceptions.
- `assert_(expr[,msg])`
  Assert checks whether the specified expression executes correctly. If not, the test fails and `msg` or `None` is returned.
- `failUnlessEqual()`
  This testing method is equivalent to `assertEqual()`.

- `failUnless(expr[,msg])`
  This method is equivalent to `assert_(expr[,msg])`.
- `failif()`
  This is the opposite to `failUnless()`.
- `fail([msg])`
  Fails the running test without any evaluation. This is commonly used when testing various possible execution paths at once and you would like to signify a failure if an improper path was taken.

▷ Line 6–10: This method tests the `title` attribute of the `Sample` class. The first test should be of course that the attribute exists and has the expected initial value (line 8). Then the title attribute is changed and we check whether the value was really stored. This might seem like overkill, but later you might change the title in a way that it uses properties instead. Then it becomes very important to check whether this test still passes.

▷ Line 12–16: First we simply check that `getDescription()` returns the correct default value. Since we do not want to use other API calls like `setDescription()` we set a new value of the description via the implementation-internal `_description` attribute (line 15). This is okay! Unit tests can make use of implementation-specific attributes and methods. Finally we just check that the correct value is returned.

▷ Line 18–25: On line 21–24 it is checked that both regular and unicode strings are set correctly. In the last line of the test we make sure that no other type of objects can be set as a description and that an error is raised.

▷ Line 28–31: This method returns a test suite that includes all test cases created in this module. It is used by the Zope 3 test runner when it picks up all available tests. You would basically add the line `unittest.makeSuite(TestCaseClass)` for each additional test case.

▷ Line 33–34: In order to make the test module runnable by itself, you can execute `unittest.main()` when the module is run.

41.3 Running the Tests

You can run the test by simply calling `pythontest_sample.py` from the directory you saved the file in. Here is the result you should see:

```
...
----------------------------------------------------------------------
```

```
Ran 3 tests in 0.001s

OK
```

The three dots represent the three tests that were run. If a test had failed, it would have been reported pointing out the failing test and providing a small traceback.

When using the default Zope 3 test runner, tests will be picked up as long as they follow some conventions.

- The tests must either be in a package or be a module called `tests`.
- If `tests` is a package, then all test modules inside must also have a name starting with `test`, as it is the case with our name `test_sample.py`.
- The test module must be somewhere in the Zope 3 source tree, since the test runner looks only for files there.

In our case, you could simply create a `tests` package in `ZOPE3/src` (do not forget the `__init__.py` file). Then place the `test_sample.py` file into this directory.

You you can use the test runner to run only the sample tests as follows from the Zope 3 root directory:

```
python test.py -vp tests.test_sample
```

The `-v` option stands for verbose mode, so that detailed information about a test failure is provided. The `-p` option enables a progress bar that tells you how many tests out of all have been completed. There are many more options that can be specified. You can get a full list of them with the option `-h`: `pythontest.py-h`.

The output of the call above is as follows:

```
Configuration file found.
Running UNIT tests at level 1
Running UNIT tests from /opt/zope/Zope3
   3/3 (100.0%): test_title (tests.test_sample.SampleTest)
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK
Running FUNCTIONAL tests at level 1
Running FUNCTIONAL tests from /opt/zope/Zope3

----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

▷ Line 1: The test runner uses a configuration file for some setup. This allows developers to use the test runner for other projects as well. This message simply tells us that the configuration file was found.

- Line 2–8: The unit tests are run. On line 4 you can see the progress bar.
- Line 9–15: The functional tests are run, since the default test runner runs both types of tests. Since we do not have any functional tests in the specified module, there are no tests to run. To just run the unit tests, use option `-u` and `-f` for just running the functional tests. See "Writing Functional Tests" for more detials on functional tests.

Exercises

1. It is not very common to do the setup – in our case `sample=Sample()` – in every test method. Instead there exists a method called `setUp()` and its counterpart `tearDown` that are run before and after each test, respectively. Change the test code above, so that it uses the `setUp()` method. In later chapters and the rest of the book we will frequently use this method of setting up tests.

2. Currently the `test_setDescription()` test only verifies that None is not allowed as input value.

   (a) Improve the test, so that all other builtin types are tested as well.

   (b) Also, make sure that any objects inheriting from `str` or `unicode` pass as valid values.

CHAPTER 42

DOCTESTS: EXAMPLE-DRIVEN UNIT TESTS

Difficulty

Newcomer

Skills

- You should have read the previous chapter on unit tests, since this chapter heavily depends the work done there.

Problem/Task

Unit tests are nice, but they are not the best implementation of what eXtreme Programming expects of testing. Testing should also serve as documentation, a requirement that the conventional unit test module pattern does not provide. This chapter will show you an alternative way of writing unit tests that can also serve well as documentation.

Solution

Python already provides docstrings for classes and methods, which serve – like the name suggests – as documentation for the object. If you would be able to write tests in the docstrings of classes and methods and execute them during test runs, all requirements of a testing framework are fulfilled. Even better, the tests would automatically become part of the documentation. This way the documentation

reader would always see a working example of the code. Since most people learn by example, this will also speed up the learning process of the technology.

The solution to this problem are doctests, which have exactly the described behavior. If you embed Python-prompt-like sample code in the docstrings of a class and register the contained module as one having doctests, then the Python code in the docstrings is executed for testing. Each docstring will be counted as a single test.

42.1 Integrating the Doctest

So how will our example from the previous chapter change in light of docstests? First of all, you can completely get rid of the `TestSample` class. Next, add the following lines to the docstring of the `Sample` class:

```
1  Examples:
2
3    >>> sample = Sample()
4
5    Here you can see how the 'title' attribute works.
6
7    >>> print sample.title
8    None
9    >>> sample.title = 'Title'
10   >>> print sample.title
11   Title
12
13   The description is implemented using a accessor and mutator method
14
15   >>> sample.getDescription()
16   ''
17   >>> sample.setDescription('Hello World')
18   >>> sample.getDescription()
19   'Hello World'
20   >>> sample.setDescription(u'Hello World')
21   >>> sample.getDescription()
22   u'Hello World'
23
24   'setDescription()' only accepts regular and unicode strings
25
26   >>> sample.setDescription(None)
27   Traceback (most recent call last):
28     File "<stdin>", line 1, in ?
29     File "test_sample.py", line 31, in setDescription
30       assert isinstance(value, (str, unicode))
31   AssertionError
```

▷ Line 1: The double colon at this line is not mistake. In Zope 3's documentation tools we assume that all docstrings are written in structured text, a plain text format that allows to insert some markup without diminishing the readability of the text. The double colon simply signifies the beginning of a code segment.

▷ Line 5, 13 & 24: It is even possible to insert additional comments for better documentation. This way examples can be explained step by step.

Next you need to change the test suite construction to look for the doctests in the `Sample` class' docstring. To do so, import the `DocTestSuite` class and change the `test_suite()` function as follows:

```
1 from zope.testing.doctestunit import DocTestSuite
2
3 def test_suite():
4     return unittest.TestSuite((
5         DocTestSuite(),
6         ))
```

The first argument to the `DocTestSuite` constructor is a string that is the dotted Python path to the module that is to be searched for doctests. If no module is specified the current one is chosen. The constructor also takes two keyword arguments, `setUp` and `tearDown`, that specify functions that are called before and after each tests. These are the equivalent methods to the `TestCase`'s `setUp()` and `tearDown()` methods.

You can now execute the test as before using `Pythontest_sample.py`, except that `ZOPE3/src` must be in your `PYTHONPATH`. The output is the same as for unit tests.

```
.
----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
```

As you can see, the three different unit tests collapsed to one doctest.

You will have to agree that doctests are a much more natural way to test your code. However, there are a couple of issues that one should be aware of when using doctests.

42.2 Shortcomings

The most obvious problem is that if you like to test attributes and properties, there is no docstring to place the tests. This problem is usually solved by testing attributes implicitly in context of other tests and/or place their tests in the class' docstring. This solution is actually good, since attributes by themselves usually do not have much functionality, but are used in combination with methods to provide functionality.

Next, it is not easy to test for certain outputs. The prime example here is `None`, since it has no representation. The easy way around this is to make the testing statement a condition. So the statement `methodReturningNone()` which should

return `None` is tested using `methodReturningNone()isNone` which should return `True`. There are also some issues when testing statements that return output whose representation is longer than a line, since the docstring checker is not smart enough the remove the indentation white space. A good example of such objects are lists and tuples. The best solution to this problem is to use the pretty printer module, `pprint`, which always represents objects in lines no longer than 80 characters and uses nice indentation rules for clarity.

Another problematic object to test are dictionaries, since their representation might change from one time to another based on the way the entries are hashed. The simplest solution to the problem is to always convert the dictionary to a list using the `items()` method and sorting the entries. This should give you a uniquely identifiable representation of the dictionary.

Over time I have noticed that using doctests tends to make me write sloppy tests. Since I think of the tests as examples to show how the class is supposed to work, I often neglect to test for all aspects and possible situation a piece of code could come into. This problem can be solved by either writing some additional classic unit tests or by creating a special testing module that contains further doctests.

While doctests cover 98% of all test situations well, there are some tests that require heavy programming. A good example of that is a test in the internationalization support that makes sure that all XML locale files can be parsed and some of the most important data is correctly evaluated. I found that it is totally okay to use regular unit tests for these scenarios.

Still, overall I think doctests are the way to go, due to their great integration into documentation!

Exercises

1. As a matter of taste, some people like it better when each method is tested in the method docstring. Therefore, move the `getDescription` and `setDescription` tests to the methods' docstrings and make sure that all three tests pass.
2. Once you have split up the tests, you always have to setup the `sample` object over and over again. Use a `setUp()` function to setup the sample as you did for the unit tests in the previous chapter.
3. (Equivalent to excercise 2 in the previous chapter.) Currently the `test_setDescription()` test only verifies that None is not allowed as input value.
   (a) Improve the test, so that all other builtin types are tested as well.
   (b) Also, make sure that any objects inheriting from `str` or `unicode` pass as valid values.

CHAPTER 43

WRITING FUNCTIONAL TESTS

Difficulty

Newcomer

Skills

- It is good to know how Zope 3 generated forms work before reading this chapter. Optional.

Problem/Task

Unit tests cover a large part of the testing requirements listed in the eXtreme Programming literature, but are not everything. There are also integration and functional tests. While integration tests can be handled with unit tests and doctests, functional tests cannot. For this reason the Zope 3 community members developed an extension to `unittest` that handles functional tests. This package is introduced in this chapter.

Solution

Unit tests are very good for testing the functionality of a particular object in absence of the environment it will eventually live in. Integration tests build on this by testing the behavior of an object in a limited environment. Then functional tests should test the behavior of an object in a fully running system. Therefore functional tests often check the user interface behavior and it is not surprising that they are found in the `browser` packages of Zope 3. In fact, in Zope 3's implementation of functional tests there exists a base test case

class for each view type, such as `zope.testing.functional.BrowserTestCase` or `zope.app.dav.ftests.dav.DAVTestCase`.

43.1 The Browser Test Case

Each custom functional test case class provides some valuable methods that help us write the tests in a fast and efficient manner. Here are the methods that the `BrowserTestCase` class provides.

- `getRootFolder()`
  Returns the root folder of the database. This method is available in every functional test case class.

- `makeRequest(path='',basic=None,form=None,env={},outstream=None)`
  This class creates a new `BrowserRequest` instance that can be used for publishing a request with the Zope publisher.

  - `path` – This is the absolute path of the URL (i.e. the URL minus the protocol, server and port) of the object that is beeing accessed.
  - `basic` – It provides the authentication information of the format "⟨`login`⟩:⟨`password`⟩". When Zope 3 is brought up for functional testing, a user with the login "mgr" and the password "mgrpw" is automatically created having the role "zope.Manager" assigned to it. So usually we will use `"mgr:mgrpw"` as our basic argument.
  - `form` – The argument is a dictionary that contains all fields that would be provided by an HTML form. Note that we should have covnerted the data already to their native Python format; be sure to only use unicode for strings.
  - `env` – This variable is also a dictionary where we can specify further environment variables, like HTTP headers. For example, the header `X-Header:value` would be an entry of the form `'HTTP_X_HEADER':value` in the dictionary.
  - `outstream` – Optionally we can define the the stream to which the outputted HTML is sent. If we do not specify one, one will be created for us.

  However, one would often not use this method directly, since it does not actually publish the request. Use the `publish()` method described below.

- `publish(self,path,basic=None,form=None,env={},handle_errors=False)`
  The method creates a request as described above, that is then published with

a fully-running Zope 3 instance and finally returns a regular browser response object that is enhanced by a couple of methods:

  - `getOutput()` – Returns all of the text that was pushed to the outstream.
  - `getBody()` – Only returns all of the HTML of the response. It therefore excludes HTTP headers.
  - `getPath()` – Returns the path that was passed to the request.

  The `path`, `basic`, `form` and `env` have the same semantics as the equally-named arguments to `makeRequest()`. If `handle_errors` is `False`, then occuring exceptions are not caught. If `True`, the default view of an exception is used and a nice formatted HTML page will be returned. As you can imagine the first option is often more useful for testing.

- `checkForBrokenLinks(body,path,basic=None)`
  Given an output body and a published path, this method checks whether the contained HTML contains any links and checks that these links are not broken. Since the availability of pages and therefore links depends on the permissions of the user, one might want to specify a login/password pair in the `basic` argument. For example, if we have published a request as a manager, it will be very likely that the returned HTML contains links that require the manager role.

43.2 Testing "ZPT Page" Views

Okay, now that we know how the `BrowserTestCase` extends the normal `unittest.TestCase`, we can use it to write some functional tests for the "add", "edit" and "index" view of the "ZPT Page" content type.

Anywhere, create a file called `test_zptpage.py` and add the following functional testing code:

```
1  import time
2  import unittest
3
4  from transaction import get_transaction
5  from zope.app.tests.functional import BrowserTestCase
6  from zope.app.zptpage.zptpage import ZPTPage
7
8  class ZPTPageTests(BrowserTestCase):
9      """Funcional tests for ZPT Page."""
10
11     template = u'''\
12     <html>
13       <body>
14         <h1 tal:content="modules/time/asctime" />
```

```
15        </body>
16      </html>'''
17
18      template2 = u'''\
19      <html>
20        <body>
21          <h1 tal:content="modules/time/asctime">time</h1>
22        </body>
23      </html>'''
24
25      def createPage(self):
26          root = self.getRootFolder()
27          root['zptpage'] = ZPTPage()
28          root['zptpage'].setSource(self.template, 'text/html')
29          get_transaction().commit()
30
31      def test_add(self):
32          response = self.publish(
33              "/+/zope.app.zptpage.ZPTPage=",
34              basic='mgr:mgrpw',
35              form={'add_input_name' : u'newzptpage',
36                    'field.expand.used' : u'',
37                    'field.source' : self.template,
38                    'field.evaluateInlineCode.used' : u'',
39                    'field.evaluateInlineCode' : u'on',
40                    'UPDATE_SUBMIT' : 'Add'})
41
42          self.assertEqual(response.getStatus(), 302)
43          self.assertEqual(response.getHeader('Location'),
44                           'http://localhost/@@contents.html')
45
46          zpt = self.getRootFolder()['newzptpage']
47          self.assertEqual(zpt.getSource(), self.template)
48          self.assertEqual(zpt.evaluateInlineCode, True)
49
50      def test_editCode(self):
51          self.createPage()
52          response = self.publish(
53              "/zptpage/@@edit.html",
54              basic='mgr:mgrpw',
55              form={'field.expand.used' : u'',
56                    'field.source' : self.template2,
57                    'UPDATE_SUBMIT' : 'Change'})
58          self.assertEqual(response.getStatus(), 200)
59          self.assert_('&gt;time&lt;' in response.getBody())
60          zpt = self.getRootFolder()['zptpage']
61          self.assertEqual(zpt.getSource(), self.template2)
62          self.checkForBrokenLinks(response.getBody(), response.getPath(),
63                                   'mgr:mgrpw')
64
65      def test_index(self):
66          self.createPage()
67          t = time.asctime()
68          response = self.publish("/zptpage", basic='mgr:mgrpw')
69          self.assertEqual(response.getStatus(), 200)
70          self.assert_(response.getBody().find('<h1>'+t+'</h1>') != -1)
71
72  def test_suite():
```

```
73     return unittest.TestSuite((
74         unittest.makeSuite(ZPTPageTests),
75         ))
76
77 if __name__=='__main__':
78     unittest.main(defaultTest='test_suite')
```

▷ Line 25–29: This is the perfect example of a helper method often used in Zope's functional tests. It creates a "ZPT Page" content object called `zptpage`. To write the new object to the ZODB, we have to commit the transaction using `get_transaction().commit()`.

▷ Line 31–48: To understand this test completely, it is surely helpful to be familiar with the way Zope 3 adds new objects and how the widgets create an HTML form. The "+"-sign in the URL is the adding view for a folder. The path that follows is simply the factory id of the content type (line 33). Instead of the factory id, we sometimes also find the name of the object's add form there.

The form dictionary is another piece of information that must be carefully constructed. First of all, the `field.expand.used` and `field.evaluateInlineCode.used` are required, whether we want to activate `expand` and `evaluateInlineCode` or not. It is required by the corresponding widgets. The `add_input_name` key contains the name the content object will recieve and `UPDATE_SUBMIT` just tells the form generator that the form was actually submitted and action should be taken. Also note that all form entries representing a field have a "field." prefix, which is done by the widgets. How did I know all these variable names? Parallel to writing the functional test, I just created a "ZPT Page" on the browser, looking at the HTML source for the names and values. There is no way I would have remembered all this!

On line 42, we check whether the request was successful. Code 302 signalizes a redirect and on line 43–44 we check that we are redirected to the correct page.

Now, it is time to check in the ZODB whether the object has really been created and that all data was set correctly. On line 46 we retrieve the object itself and consequently we check that the source is set correctly and the `evaluateInlineCode` flag was turned on (line 48) as the request demanded in the form (line 39).

▷ Line 50–63: Before we can test whether the data of a "ZPT Page" can be edited correctly, we have to create one. Here the `createPage()` method comes in handy, which quickly creates a page that we can use. Having done previous test already, the contents of the `form` dictionary should be obvious.

Since the edit page returns itself, the status of the response should be 200. We also inspect the body of the response to make sure that the temlpate was stored correctly.

One extremly useful feature of the `BrowserTestCase` is the check for broken links in the returned page. I would suggest that you do this test whenever a HTML page is returned by the response.

▷ Line 65–70: Here we simply test the default view of the "ZPT Page". No complicated forms or environments are necessary. We just need to make sure that the template is executed correctly.

43.3 Running Functional Tests

The testing code directly depends on the Zope 3 source tree, so make sure to have it in your Python path. In Un*x/Linux we can do this using

```
export PYTHONPATH=$PYTHONPATH:ZOPE3/src
```

where `ZOPE3` is the path to our Zope 3 installation. Furthermore, functional tests depend on finding a file called `ftesting.zcml`, which is used to bring up the Zope 3 application server. Therefore it is best to just go to the directory `ZOPE3`, since there exists such a file. You can now execute our new funtional tests using

```
python path/to/ftest/test_zptpage.py
```

You will notice that these tests will take a couple seconds (5-10 seconds) to run. This is okay, since the functional tests have to bring up the entire Zope 3 system, which by itself will take about 4-10 seconds.

```
...
----------------------------------------------------------------------
Ran 3 tests in 16.623s

OK
```

As usual you also use the test runner to execute the tests.

Exercises

1. Add another functional test that checks the “Preview” and “Inline Code” screen.

CHAPTER 44

CREATING FUNCTIONAL DOCTESTS

Difficulty

Sprinter

Skills

- You should be familiar on how to start Zope 3 and use the Web UI.

Problem/Task

When writing functional tests, as we have done in the previous chapter, we first have to use the Web UI to see what and how we want to test. We also often need to look at the HTML code to determine all important form data elements. Then we start to write some Python code to mimic the behavior, which is often very frustrating and tedious, to say the least. Wouldn't it be nice, if some mechanism would record our actions and then simply convert the recorded session to a test that we can simply comment on? This chapter will tell you exactly how to do this using functional doctests.

Solution

In the previous chapter we developed some functional tests for the common tasks of a "ZPT Page", creating the component, edit the content, and finally calling the default view to render the page. In this chapter we will recreate only the rendering of the template for simplicity.

Creating functional doctests requires some specific setup of Zope 3 and a nice Python script called `tcpwatch.py` by Shane Hathaway. TCP Watch will record

the HTTP requests and responses for us, which we will use to create the functional tests. Next a script called `dochttp.py` is used to convert the TCP Watch output to a functional doctest, which you can then document and adjust as you desire.

44.1 Step I: Setting up the Zope 3 Environment

The best way to run a recording session is to have a clean ZODB. Therefore, save your old `Data.fs` as `Data.fs.orig` and remove `Data.fs`. This way you start at the same position as the functional testing framework. Also, the functional tests know only about one user with login "mgr" and password "mgrpw". The user has the "zope.Manager" role. You will need to setup this user, if you want to be able to access all screens. The user is easily added by placing the following two directives to `principals.zcml`:

```
<principal
    id="zope.mgr"
    title="Manager"
    login="mgr"
    password="mgrpw" />

<grant role="zope.Manager" principal="zope.mgr" />
```

In fact, I just simply copied this code from the `ftesting.zcml` file, which is actually used when running the functional tests.

Now simply start Zope 3. I assume for the rest of this chapter that Zope runs on port 8080. Since we only want to test the rendering of the ZPT, add a a new "ZPT Page" via the Web GUI having the following template code:

```
tml>
<body>
  <h1 tal:content="modules/time/asctime" />
</body>
html>
```

44.2 Step II: Setting up TCP Watch

As mentioned before, TCP Watch is used to record a Web GUI session. The distribution of the script can be found on Shane's Web site, `http://hathawaymix.org/Software/TCPWatch`. Download the latest version.

Once the download is complete, untar the archive.

```
tar xvzf tcpwatch-1.3.tar.gz
```

Now enter the newly created directory `tcpwatch`. You can now install the script by calling

```
python setup.py install
```

You might have to be root to call this command, since Python might be installed in a directory you do not have write access to.

Now create a temporary directory that can be used by TCP Watch to store the collected requests and responses. The easiest will be

```
mkdir tmp
```

Start the script using

```
/path/to/python/bin/tcpwatch.py -L 8081:8080 -s -r tmp
```

The `-L` option tells TCP Watch to listen on port 8081 for incoming connections, `-s` outputs the result to `stdout` instead of a graphical window, and `-r`⟨`dir`⟩ specifies the directory to record the session.

Once started you can access Zope also via port 8081, except that all communication between the client and server is reported by TPC Watch.

44.3 Step III: Recording a Session

Now that everything is setup, you can just do whatever you want via port 8081. Don't forget to log in as "mgr" on port 8081, so that the authentication will also work via the functional test. In our case, we just call the URL `http://localhost:8081/newzptpage`, which will render the page output the result, all of which is recorded.

Once you are done recording, shut down TPC Watch using CTRL-C. You may also shut down Zope 3.

44.4 Step IV: Creating and Running the Test

Once the session is recorded, you can convert it to a functional test using

```
Python ZOPE3/src/zope/app/tests/dochttp.py ./tmp > zptpage_raw.txt
```

I prefer to store the output of this script in a temporary file and copy request for request into the final text file. The raw functional test should look like this:

```
>>> print http(r"""
... GET /newzptpage HTTP/1.1
... Cache-Control: no-cache
... Pragma: no-cache
... """)
HTTP/1.1 200 Ok
Content-Length: 72
Content-Type: text/html;charset=utf-8
<BLANKLINE>
<html>
```

```
    <body>
      <h1>Thu Aug 26 12:24:26 2004</h1>
    </body>
  </html>
  <BLANKLINE>
```

Our final functional test will be stored in `zptpage.txt` though. Since we added the `newzptpage` object before the recording session, our final test file must add this object via Python code. Here is the final version of `zptpage.txt`:

```
========
ZPT Page
========

This file demonstrates how a page template is rendered in content
space. Before we can render the page though, we have to create one. The
template content will be:

  >>> template = u'''\
  ... <html>
  ...   <body>
  ...     <h1 tal:content="modules/time/asctime" />
  ...   </body>
  ... </html>'''

Next we have to create the ZPT Page in the root folder. The root folder of the
test setup can be simply retrieved by calling `getRootFolder()`. At the end we
have to commit the transaction, so that the page will be stored in the ZODB
and available for requests to be accessed.

  >>> from transaction import get_transaction
  >>> from zope.app.zptpage.zptpage import ZPTPage
  >>> root = getRootFolder()
  >>> root['newzptpage'] = ZPTPage()
  >>> root['newzptpage'].setSource(template, 'text/html')
  >>> get_transaction().commit()

Now that we have the page setup, we can just send the HTTP request by calling
the function `http()`, which will return the complete HTTP response. When
comparing the

  >>> print http(r"""
  ... GET /newzptpage HTTP/1.1
  ... Cache-Control: no-cache
  ... Pragma: no-cache
  ... """)
  HTTP/1.1 200 Ok
  Content-Length: 72
  Content-Type: text/html;charset=utf-8
  <BLANKLINE>
  <html>
    <body>
      <h1>...</h1>
    </body>
  </html>
  <BLANKLINE>
```

- Line 21–26: Here we add a "ZPT Page" named "newzptpage" to the root folder. This is identical to the method we created in the previous chapter.
- Line 43: Here I replaced the date/time string with three dots (...), which means that the content is variable. This is necessary, since the session and the functional test are run at different times.

Now that the test is written we make it runnable with a usual setup. In a file named `ftests.py` add the following test setup.

```
1 import unittest
2 from zope.app.tests.functional import FunctionalDocFileSuite
3
4 def test_suite():
5     return FunctionalDocFileSuite('zptpage.txt')
6
7 if __name__ == '__main__':
8     unittest.main(defaultTest='test_suite')
```

When running a functional doctest, you just use the `FunctionalDocFileSuite` class to create a doctest suite. You are now able to run the tests with the test runner as usual.

CHAPTER 45

WRITING TESTS AGAINST INTERFACES

Difficulty

Newcomer

Skills

- You should be familiar with Python interfaces. If necessary, read the "An Introduction to Interfaces" chapter.
- You should know about the `unittest` package, especially the material covered in the "Writing Basic Unit Tests" chapter.

Problem/Task

When one expects an interface to be implemented multiple times, it is good to provide a set of generic tests that verify the correct semantics of this interface. In Zope 3 we refer to these abstract tests as "interface tests". This chapter will describe how to implement and use such tests using two different implementations of a simple interface.

Solution

45.1 Introduction

In Zope 3 we have many interfaces that we expect to be implemented multiple times. The prime example is the `IContainer` interface, which is primarily implemented by

`Folder`, but also by many other objects that contain some other content. Here it would be useful to implement some set of tests which verify that `Folder` and other classes correctly implement `IContainer`.

Interface tests are abstract tests – i.e. they do not run by themselves, since they do not know about any implementation – that provide a set of common tests. The advantage of these tests is that the implementor of this interface immediately has some feedback about his implementation of the interface. However, one should not mistake the interface tests to be a replacement of a complete set of unit tests, but rather as a supplement. Interface tests by definition cannot test implementation details, something that is required of unit tests. Additional tests also ensure a higher quality of code.

There are a couple of characteristics that you will be able to recognize in any interface test. First, an interface should always have a test that verifies that the interface implementation fulfills the contract. This can be done using the `verifyObject(interface,instance)` method, which is found at `zope.interface.verify`. Second, while the interface test is abstract, it needs to get an instance of the implementation from somewhere. For this reason an interface test should always provide an abstract method that returns an instance of the object. By convention this method is called `makeTestObject` and it should look like that:

```
1 def makeTestObject(self):
2     raise NotImplemented()
```

Each test case that inherits from the interface test should then realize this method by returning an instance of the object to be tested.

But how can we determine what should be part of an interface test? The best way to approach the problem is by thinking about the functionality that the attributes and methods of the interface provide. You may also ask about its behavior inside the system? Interface tests often model actual usages of an object, while implementation tests also cover a lot of corner cases and exceptions, something that is often hard to do with interface tests, since you are bound to the interface-declared methods and attributes. From another point of view, since tests should document an object, think of interface tests as documentation on how the interface should be used and behave normally.

45.2 The `ISample` Interface, Its Tests, and Its Implementations

Reusing the examples from the unit and doc test chapters, we develop an `ISample` interface that provides a `title` attribute and uses the methods `getDescription` and `setDescription` for dealing with the description of the object.

Again, we would like to keep the code contained in one file for simplicity, so open a file `test_isample.py` anywhere and add the following interface to it:

```
1  from zope.interface import implements, Interface, Attribute
2
3  class ISample(Interface):
4      """This is a Sample."""
5
6      title = Attribute('The title of the sample')
7
8      def setDescription(value):
9          """Set the description of the Sample.
10
11         Only regular and unicode values should be accepted.
12         """
13
14     def getDescription():
15         """Return the value of the description."""
```

I assume you know about interfaces, so there is nothing interesting here. The next step is to write the interface tests, so add the following `TestCase` class. You will notice how similar these tests are to the ones developed before.

```
1  import unittest
2  from zope.interface.verify import verifyObject
3
4  class TestISample(unittest.TestCase):
5      """Test the ISample interface"""
6
7      def makeTestObject(self):
8          """Returns an ISample instance"""
9          raise NotImplemented()
10
11     def test_verifyInterfaceImplementation(self):
12         self.assert_(verifyObject(ISample, self.makeTestObject()))
13
14     def test_title(self):
15         sample = self.makeTestObject()
16         self.assertEqual(sample.title, None)
17         sample.title = 'Sample Title'
18         self.assertEqual(sample.title, 'Sample Title')
19
20     def test_setgetDescription(self):
21         sample = self.makeTestObject()
22         self.assertEqual(sample.getDescription(), '')
23         sample.setDescription('Description')
24         self.assertEqual(sample.getDescription(), 'Description')
25         self.assertRaises(AssertionError, sample.setDescription, None)
```

▷ Line 4–6: Here is the promised method to create object instances.

▷ Line 8–9: As mentioned before, every interface test case should check whether the object implements the tested interface. It is the easiest test you will ever write, and it is one of the most important ones.

▷ Line 11–15: This test is equivalent to the test we wrote before, except that we do not create the sample instance by using the class, but using some indirection by asking the `makeTestObject()` to create one for us.

▷ Line 17–22: In interface tests it does not make much sense to test the accessor and mutator method of a particular attribute seperately, since you do not know how the data is stored anyway. So, similar to the test before, we test some combinations of calling the description getter and setter.

Now that we have an interface and some tests for it, we are ready to create an implementation. In fact, we will create two, so that you can see the independence of the interface tests to specific implementations.

The first implementation is equivalent to the one we used in the unit test chapter, except that we call it `Sample1` now and that we tell it that it implements `ISample`.

```
1  class Sample1(object):
2      """A trivial ISample implementation."""
3
4      implements(ISample)
5
6      # See ISample
7      title = None
8
9      def __init__(self):
10         """Create objects."""
11         self._description = ''
12
13     def setDescription(self, value):
14         """See ISample"""
15         assert isinstance(value, (str, unicode))
16         self._description = value
17
18     def getDescription(self):
19         """See ISample"""
20         return self._description
```

The second implementation uses Python's `property` feature to implement its `title` attribute and uses a different attribute name, `__desc`, to store the data of the description.

```
1  class Sample2(object):
2      """A trivial ISample implementation."""
3
4      implements(ISample)
5
6      def __init__(self):
7          """Create objects."""
8          self.__desc = ''
9          self.__title = None
10
11     def getTitle(self):
12         return self.__title
13
```

```
14      def setTitle(self, value):
15          self.__title = value
16
17      def setDescription(self, value):
18          """See ISample"""
19          assert isinstance(value, (str, unicode))
20          self.__desc = value
21
22      def getDescription(self):
23          """See ISample"""
24          return self.__desc
25
26      description = property(getDescription, setDescription)
27
28      # See ISample
29      title = property(getTitle, setTitle)
```

▷ Line 26: While this implementation chooses to provide a convenience property to `setDescription` and `getDescription`, it is not part of the interface and should not be tested in the interface tests. However, the specific implementation tests should cover this feature.

These two implementations are different enough that the interface tests should fail, if we would have included implementation-specific testing code. The tests can now be implemented quickly:

```
1  class TestSample1(TestISample):
2
3      def makeTestObject(self):
4          return Sample1()
5
6      # Sample1-specific tests are here
7
8
9  class TestSample2(TestISample):
10
11     def makeTestObject(self):
12         return Sample2()
13
14     # Sample2-specific tests are here
15
16
17 def test_suite():
18     return unittest.TestSuite((
19         unittest.makeSuite(TestSample1),
20         unittest.makeSuite(TestSample2)
21         ))
22
23 if __name__ == '__main__':
24     unittest.main(defaultTest='test_suite')
```

▷ Line 1–6 & 9–14: The realization of the `TestISample` tests is easily done by implementing the `makeTestObject()` method. We did not write any implementation-specific test in order to keep the code snippets small and concise.

▷ Line 12–19: This is just the usual test environment boiler plate.

To run the tests, you need to make sure to have ⟨ZOPE3⟩/src in your PYTHONPATH, since this code depends on zope.interface. Then you can simply execute the code using

```
python test_sampleiface.py
```

from the directory containing test_sampleiface.py.

```
Configuration file found.
Running UNIT tests at level 1
Running UNIT tests from /opt/zope/Zope3/Zope3-Cookbook
   6/6 (100.0%): test_verifyInterfaceImplementation (...leiface.TestSample2)
----------------------------------------------------------------------
Ran 6 tests in 0.004s

OK
```

As you can see, you just wrote three tests, but for the two implementations six tests run. Interface tests are a great way to add additional tests (that multiply quickly) and they are a great motivation to keep on writing tests, a task that can be annoying sometimes.

Exercises

1. Develop implementation-specific tests for each `ISample` implementation.
2. Change the `ISample` interface and tests in a way that it uses a `description` attribute instead of `getDescription()` and `setDescription()`. Note: The requirement that the description value can only be a regular or unicde string should still be valid.

PART VIII

Appendix

Appendix A

Glossary of Terms

Adapter

This basic component is used to provide additional functionality for an object. Formally, it uses one interface (implemented by an object) to provide (adapt to) another interface. For example, given an object `IMessage` one can develop an adapter that provides the interface `IMailSubscription` for this object.

The adapter can also be seen as some form of object wrapper that is able to use the object to provide new or additional functionality. The adapter's constructor will therefore always take one object as argument. This object is commonly known as the context.

Advanced uses also allow multi-adaptation, where several objects – each implementing a specified interface – are adapted to one interface. Therefore the adapter has several contexts.

See also: Component Architecture

Annotation

Annotations are additional, non-critical pieces of information about an object. This includes all meta data; for example the Dublin Core.

An annotations provides an object-wide data namespace and is identified by the annotation key. Some care should be taken to choose the key wisely to avoid key name collisions. A safe way is to use XML-like namespace declarations, like `http://namespaces.zope.org/mailsubscription#1.1`, but the safest way is to use the dotted name of the object that creates an annotation, like `zope.app.messageboard.MailSubscription`.

Annotations are also often used by adapters to store additional data to save its state. For example the `IMessage`'s `MailSubscription` adapter stores the subscribed E-mail addresses in an annotation of the message.

The most common form of an annotation is the attribute annotation, which simply stores the annotation in a special attribute of the object. An object simply has to implement the `IAttributeAnnotable` marker interface to allow attribute annotations.

Reference: `zope.app.annotation.interfaces.IAnnotations`, `zope.app.annotation.interfaces.AttributeAnnotatable`

Cache

The cache allows you to store results of expensive operations using a key. For example, sometimes it is not necessary to render a page template any time a request is made, since the output might not change for a long time. The most common implementation of a cache is the RAM cache, which stores the cache entries in RAM for fast access. Caches, when carefully used, can increase the performance of your application while not disturbing the dynamics of it.

Reference: `zope.app.cache.interfaces.ICache`

Checker

Checkers describe the security assertions made for a particular type (types are classes that inherit `object`). Given an object and an attribute name, the checker can tell you whether the current user has the right to access or even mutate the attribute. The checker can also create a protective proxy for a given object.

A common checker is the name-based checker, where read and write permissions are stored for each attribute name. These checkers have also the advantage that they can be easily introspected about their configuration.

Reference: `zope.security.interfaces.IChecker`
See also: Security, Proxy

Component

A component can be any Python object and represents the most fundamental concept of Zope 3. Components are commonly classes, though they can be any arbitrary Python object. The fundamental components are Service, Adapter, Utility, Presentation, Content, and Factory. One could consider Interface to be a component too, though they are usually considered utilities.

See also: Component Architecture, Service, Adapter, Utility, Presentation, Factory, Interface

Component Architecture

The component architecture is a set of five fundamental services (registries) in which the specific components are managed. The services are: Service Manager (or also known as the Service Service), Adapter Service, Utility Service, Presentation Service (also known as View Service), and Factory Service.

Components can interact through the interfaces they implement. Using interfaces and the above mentioned services, you are able build any component-based system you can image. The advantage over components versus larger mix-in based classes and convention-driving naming schemes is that components have much better defined functionality, clear interaction contracts and better reusability.

Reference: `zope.component.interface.IComponentArchitecture`
See also: Component, Service

Container

A container is an object that can contain other objects, differentiating its entries by name. It is very similar to a dictionary. In fact, the container implements the entire Python dictionary (mapping) API, so that the same syntax may be used. Any content object that contains children should implement the container interface.

One important feature of the container has become the fact that it can restrict the types of components it can contain. This is accomplished by setting pre-conditions on the `__setitem__()` method of the container interface.

Reference: `zope.app.container.interfaces.IContainer`
See also: Folder, Content

Content

This basic component stores data/content in the object database or anywhere else. Interfaces that are meant to be used by content components usually provide the `IContentType` marker interface.

Content components are usually created by factories, which are defined in the configuration. Prominent examples of content components include `File`, `Image`, `TemplatedPage`, and `SQLScript`.

Reference: `zope.app.content.interfaces.IContentType`
See also: Component

Context

The context is usually a place in the object hierarchy from which code is executed. Having components be associated with a context, we are able to provide location-specific implementations and instances of a particular component. This allows us to override, enhance or enrich the functionality of a component as we walk down a path in the object tree.

The concept of of contexts is also used for adapters and views. There they describe the component that is wrapped by the adapter. For the view the context is simply the component that the view is for.

See also: Global Components, Local Components, Adapter, View

Doctests

Doctests provide the capability of writing unit tests inside docstrings or simple text files. However, a test does not look like straight Python code, but like the screen output of an interactive Python session.

Doctests allow us to use the tests themselves as documentation in forms of examples. It is also much easier this way to document the steps the test goes through. Doctests have therefore become the primary way to write unit tests in Zope 3.

Reference: `zope.testing.doctestunit`
See also: Tests, Unit Tests, Functional Tests

Document Template Markup Language (DTML)

DTML uses custom XML tags to create templates for markup. It was the original templating technology in Zope and is still favored by many developers, since it is easy to understand and extend. DTML is available in Zope 3 as a content component and is used for dynamic SQL scripts. DTML, in contrast to ZPT, does not require the template or the output to be well-formed XML.

Reference: `zope.documenttemplate`
See also: Zope Page Template

Dublin Core

The Dublin Core defines a finite set of meta-data attributes, such as "name" and "title". For a complete list of attributes including detailed descriptions you can see the relevant chapter in the book, the interfaces, or the official Dublin Core Web site (`http://www.dublincore.org/`).

Reference: `zope.app.dublincore.interfaces.IZopeDublinCore`
See also: Annotation, Meta-data

Event

An event in the Zope 3 sense is an object that represents a taken or to be taken action of the system. An event can contain and represent anything as long as it implements `IEvent`. Some common examples include `ObjectCreatedEvent`, `ObjectModifiedEvent` and `IObjectCopiedEvent`. All these belong to the group of object events and always carry the affected object with them.

Reference: `zope.app.event.interfaces.IEvent`
See also: Event Channel, Event Subscriber

Event Channel

An event channel (or in general an `ISubscribable` object) can send events to a list of event subscribers. When a subscription is made, the `subscriber` object, an `event_type` and a `filter` is passed. The subscriber is only notified when the to-be-distributed event matches the `event_type` and the filter condition is satisfied.

Think about the event channel in terms of a mailing list manager, like Mailman. People can subscribe with an E-mail address (event subscriber) to a particular mailing list (event type). When someone sends a mail to a mailing list (which goes through the list manager), the list manager figures out to which mailing list the mail (event) should go and sends it (notifies) all subscribers.

See also: Event, Event Subscriber

Event Subscriber

An event subscriber can be subscribed to an event channel. Whenever an event, in which the subscriber is interested in, arrives the event channel, the event subscriber is notified about the event.

See also: Event, Event Channel

Factory

A factory is responsible for creating other components. Additionally one can check what interfaces will be implemented by the component the factory creates. Factories also carry a title and description, which can be used in user interfaces to represent the factory.

Note that factories should *never* require a permission to be executed, since they just create an object. Just because an object can be created, it does not mean that the creator has any permissions to access anything in the object.

Reference: `zope.component.interfaces.IFactory`
See also: Component, Component Architecture

Field

Fields are extensions to interface `Attributes` that describe an object's attribute in much more detail, since a lot of meta data about the attribute is supplied. Some fields that are available are: Text, TextLine, Int, Float, Bool, Choice, Tuple, List, Set, Dict, Date, Time, Datetime, Bytes, BytesLine, and Password.

Fields are used to validate attribute values and to create input elements in user interfaces.

Reference: `zope.schema.interfaces`
See also: Schema, Form, Widget

Folder

The folder is used in content space as the primary container-like object. It places no restrictions on the type of object it can contain. Furthermore, any folder can be upgraded to become a site.

Folders implement the `IContainer` interface, of course.

Reference: `zope.app.folder.interfaces.IFolder`
See also: Container, Content, Site

Form

Practically, a form is the view of a schema or parts of a schema. For the browser, the form is able to produce a full HTML form whose data input is converted to a native Python object, validated and finally stored in the component that implements the schema. Forms, in combination with schemas, are Zope 3's solution for autogenerated input and display GUIs that are solely based on information provided by the component.

See also: Schema, Field, Widget, View

Functional Tests

This class of tests run under a fully functional system. Often we use them to test whether views are generated and handle data correctly and whether their interaction with low-level components works without failure.

Functional tests are usually specific to a particular presentation type, such as the Web browser or FTP.

Reference: `zope.app.tests.functional`
See also: Tests

Global Components

Any component that is created without having a context is considered global and will be always available. Commonly, global components are created during startup, mainly through ZCML directives.

Global components cannot store a state between Zope startups. Whenever Zope is shut down all of the component is destroyed. Therefore, the ZCML directives completely describe the state of such components. Care should be taken that no other mechanism can modify their state.

See also: Component, Local Component, Zope Configuration Markup Language

Interaction

The interaction decides whether all involved participation (acting principals) have the necessary permission required to access an object's attribute. The interaction is the heart of the security implementation, since it applies the rules of the system on particular actions.

Reference: `zope.security.interfaces.IInteraction`
See also: Security, Security Policy, Participation, Principal

Interface

An interface in Zope, like in many other programming languages and environments, describes the functionality of an object in a formal way. It specifies all methods and attributes of an object that are part of the public API. Interfaces are also used as the primary API documentation.

Reference: `zope.interface.interfaces.IInterface`
See also: Component, Component Architecture

Internationalization

Internationalization, commonly abbreviated as I18n, is the process of making a software adjustable to different regions and languages. This is not just about translating text, but also allowing different number and date/time formats, unicode support, text encoding/decoding and so on. Zope 3 is completely internationalized.

See also: Localization, Locale

Local Components

These are components that are only available in a relative context or place. They are defined in Site objects (special folders) and will be available in the site and its children. The creation and configuration is commonly accomplished via the Web-based GUI (ZMI).

Local Components can store a state across Zope startups, since they are stored in the ZODB.

See also: Component, Global Component, Site, Context

Locale

A locale is an object that contains specific information about a user's region and language, such as number and date/time formats, month names, and so on. Zope 3 uses LDML-based XML locale files to get its data for over 200 locales. See `http://www.openi18n.org` for more information.

Reference: `zope.i18n.interfaces.locale.ILocale`
See also: Internationalization, Localization

Localization

Localization, commonly abbreviated as L10n, is the actual process of making a software available for a particular region and language. Since the region information is usually available via the locale, the L10n process for Zope 3 consists mainly of translating message strings from the standard message catalog.

See also: Internationalization, Locale, Message Catalog

Location

A location is a place along the object tree. Objects, that support themselves to be placed into a location, have information about its parent and the name through

which it is available from the parent. A common example of components that are commonly given a location are content components.

Obviously, not all components must have a location, such as all global components.

Reference: `zope.app.location.interfaces.ILocation`
See also: Local Components, Global Components, Content

Message Catalog

Message catalogs are collections of translations for a particular language and domain. For filesystem-based code, the standard Gettext message catalog format (also known as PO files) and directory structure is used, while the local version makes use of advanced Python structures.

Reference: `zope.i18n.interfaces.IMessageCatalog`
See also: Localization, Domain

Meta-data

Meta-data in Zope 3 is data that is additional data about an object. It usually allows the object to be better integrated in its environment without convoluting the object's original data namespace. Examples of meta-data items include "title", "author", "size" and "modification data" of an object.

See also: Annotation, Dublin Core

Namespace

In Zope 3 this term is used in two different ways. When dealing with XML (like ZCML or ZPT), the term namespace is used to refer to XML namespaces, which, for example, play a vital role by providing new directives in ZCML.

The other use of namespace is in traversable URLs. Whenever a path segment starts and ends with "++" a new traversal namespace is accessed. Traversal namespaces are primarily used to separate content from software from documentation, but also for inserting new parameters, like the skin to be used or the virtual hosting URL. Available namespaces include: etc, view, resource, attribute, item, acquire, skin, help, vh, and apidoc.

Reference: `zope.app.traversing.namespace`
See also: Zope Configuration Markup Language, Traversal

Pair Programming

When pair programming, two developers sit at one computer and develop together a piece of software. The idea is that the driver (the programmer doing the typing) is constantly checked by the second person for typos, bugs and design flaws. Pair programming also accelerates when design decisions need to be made, since there is immediate feedback of ideas.

See also: Sprint

Participation

The participation, which would have been better named participant, represents one participating party (principal) in the interaction. The participation consists basically of a principal-interaction pair.

Reference: `zope.security.interfaces.IParticipation`
See also: Security, Principal, Interaction, Security Policy

Permission

Permissions are used to allow or deny a user access to an object's attribute. They represent the smallest unit in the access control list. Permissions are just common strings, except for `zope.security.checker.CheckerPublic` which is the permission that makes an attribute available to everyone (the public).

See also: Security, Checker, Role

Persistent

Objects that are considered "persistent" can be stored in the ZODB and attribute mutations are automatically registered and stored as well. Objects that want to be persistent must inherit `persistent.Persistent` or provide another implementation of the persistent interface.

Reference: `persistent.interfaces.IPersistent`
See also: Zope Object Database

Presentation

Presentation components provide an interface between the system-internal components and the user interface or other communication protocol. This includes Browser,

WebDAV, XML-RPC and FTP. However, the output can be anything one could image. In this sense, presentation components are like adapters, except that they commonly adapt to external interfaces instead of formal Python/Zope ones.

If a presentation component does not create a publishable presentation output (i.e. just an HTML snippet instead of an HTML document), then it can also provide a formal interface. These type of presentations are then used by other presentation components. A prime example are widgets, which provide view snippets of fields.

Reference: `zope.component.interfaces.IPresentation`
See also: Component, Component Architecture, View, Resource

Principal

In general, a principal is an agent using the system. The system can associate permissions with a principal and therefore grant access to the objects that require the permissions. Principals can be security certificates, groups, and most commonly users.

Reference: `zope.app.security.interfaces.IPrincipal`
See also: Security, Permission, User

Proxy

Proxies, in general, are object wrappers that either protect objects or add additional functionality to them. In Zope, however, the main proxy class is defined in the security package and is responsible for providing a protective layer around the object, so that only principals with sufficient permissions can access the attributes of the object.

Reference: `zope.security.proxy.Proxy`
See also: Security, Checker

Publisher

The Zope publisher is responsible for publishing a request in the Zope application. Thereby it relies heavily on the request's information to find the object, handle output and even know about the principal that sent the request. Most of the publishing process, however, is delegated to other components.

Reference: `zope.publisher.interfaces.IPublisher`
See also: Request, Principal, Component

Relational Database Adapter

A database adapter is a connection from Zope to a specific relational database. For each existing relational database there is a slightly different database adapter. Each instance of an adapter connects to one particular database.

Reference: `zope.app.rdb.interfaces.IZopeDatabaseAdapter`

Python Developer

This term is used to refer to the audience that will develop filesystem-based Python packages and products for Zope 3. Python Developers are the most advanced group of developers. It is expected that they know Python very well and are familiar with common programming patterns and formal object-oriented development.

Request

A request contains all the information the system will know about a user's inquiry to the publisher. Information the request carries include the path to the accessed object, the user, the user's region and language, the output format of the returned data, possible environment variables, and input data.

Reference: `zope.publisher.interfaces.IRequest`
See also: Publisher, User

Resource

A resource is a presentation component that does not depend on another component. It is used to provide context insensitive data. Most commonly, browser-specific resources are used to provide CSS, Javascript and picture files for an HTML page.

Reference: `zope.component.interfaces.IResource`
See also: Component, Presentation, View

Role

Roles are collections of permissions that can be granted to a principal. They are provided by the standard Zope security policy, which is also responsible for the management of the roles.

Note that the role is a concept that must not be provided by all security policies and therefore application code should not depend on them.

Reference: `zope.products.securitypolicy.interfaces.IRole`

See also: Security, Permission, Principal

Schema

A schema is an interfaces that contains fields instead of methods and attributes. Schemas are used to provide additional meta-data for the fields it supports. This additional data helps the system to validate values and autogenerate user interfaces using widgets, such as HTML forms.

See also: Field, Form, Widget

Scripter

This audience has a classic HTML, CSS and Javascript development background. They are using Zope to develop dynamic Web pages as easily as possible. They are not familiar with any programming patterns and formal development. They just want to get the job done!

Zope 3 tries to provide facilities for this group by allowing content-space templating and high-level TTW development of components. We also intend to provide migration paths for the scripter to become a real Zope 3 developer.

Security

Zope has a well-designed security model that allows to protect its components in untrusted environments. Untrusted environments are marked by uncontrollable input and object access. Whenever a component is requested by untrusted code, it is put in a spacesuit, the proxy. When the untrusted code requests an attribute the proxy checks with the object's checker whether the registered user for the request has the necessary permission to access the attribute.

The decision process whether a principal has the required permission for an object's attribute is up to the security to decide.

See also: Checker, Permission, Principal, Proxy, Role, Security Policy, Interaction, Participation

Security Policy

The security policy is the blueprint for the interaction. Its only responsibility is to create an interaction, given a set of participations. While roles are not necessary for the security to function, the default Zope security policy provides extensive facilities to manage and use roles in its decision process.

Reference: `zope.security.interfaces.ISecurityPolicy`
See also: Security, Interaction, Participation, Principal, Permission, Role

Service

A service provides some fundamental functionality to the system and its existence is necessary for the system's correct functioning. Services, unlike many other components, do not self-destruct or are created every time they are being called. Therefore it is possible for them to have some state. Global services are always completely built up from scratch getting all its data from the configuration process, whereby local implementations can store the state in the ZODB and are therefore saved over any amount of runtimes.

See also: Component, Component Architecture

Session

A session allows to store a user's state in an application. This is only important, if the connection to the user is closed after every request, meaning that the state would usually be lost. HTTP is a protocol that closes connections after each request, for example.

Reference: `zope.app.session.interfaces.ISession`

Site

A site is a folder that can also contain software and configuration. It provides a connection from content space to local software space and allows the development of through-the-web components. From another point of view, a site simply provides a local service manager.

Folders can always be converted to sites.

Reference: `zope.app.site.interfaces.ISite`
See also: Component, Component Architecture, Service, Folder

Sprint

A sprint in general is a two to three day session of intensive software development. Since the idea stems from eXtreme Programming, hacking is not an option, but instead disciplined pair programming, testing and documenting is asked for. Sprints were used during the Zope 3 development to boost development and introduce the software to interested parties.

See also: Pair Programming

Template Attribute Language (TAL)

This extension to HTML allows us to do server-side scripting and templating without using non-standard HTML. Since there exists valid HTML at any time, WYSIWIG tools like Dreamweaver can be used to edit the HTML without disturbing the template scripting.

Reference: `zope.tal`
See also: TALES, Zope Page Template

Template Attribute Language Expression Syntax (TALES)

TALES are expressions that evaluate code of a specified type and return its results. The various expression types are managed by the TALES engine. In a TALES expression, one can specify the expression type at the beginning. Here are some of the most common default expression types:

- "path: " – This expression takes a URL-like path and tries to evaluate (traverse) it to an object. This expression is the default one in most TALES engines.
- "string: " – Returns a string, but it has interpolation built in that handles path expressions.
- "python: " – This expression returns the result of a Python expression.
- "not: " – If the result of the following expression is a boolean, simply negate that result.
- "exists: " – Determine whether an object of a given path (as in path expression) exists.

Reference: `zope.tales.engine`
See also: TAL, Zope Page Template

Tests

Tests are meant to check software for its functionality and discover possible bugs. This programming technique was primarily pushed by eXtreme Programming, a software development process that was used to develop Zope 3. There are several levels of testing: unit, regression, and functional tests.

See also: Unit Tests, Functional Tests

Through-the-Web Development

This term, commonly abbreviated TTW development, refers to the process of developing software via the Zope 3 Web interface (ZMI). Developing TTW is often simpler than hardcore Python product development and provides the scripter with a path to migrate to more formal, component-oriented development. TTW-developed components are also commonly known as local components, since they are only applicable for the site they were developed in.

See also: Zope Management Interface, Site, Local Component

Transaction

A transaction is a collection of actions in a database, in our case the ZODB. Zope transactions, like relational database transactions, can be begun, committed, and aborted. Upon commit, if the set of actions do not cause any problems, the actions are executed. If errors occur an exception is raised. Sometimes errors happen at different places of the system; in this case the pending list of actions can be aborted.

The process of first checking whether a commit will be successful and then doing the actual commit is known as a two-phase commit of a transaction. Two-phase commits are important for data integrity and consistency.

Reference: `ZODB.interfaces.ITransaction`
See also: Publisher, ZODB

Translation Domain

Not all words and phrases that will be ever used is Zope 3 are applicable in all applications. Domains are used to separate translations by usage. For example, all of the standard Zope 3 distribution uses the domain "zope", whereby the Zope 3 Wiki uses "zwiki".

Another use for domains is sometimes to differentiate between different meanings of a word. For example, the word "Sun" could be the abbreviation for "Sunday", our star the "Sun" or the company "Sun" (Standford University Networks). So for example, "Sun" as in "Sunday" could be in a domain called "calendar", whereby "Sun" as in our star could be in domain called "star".

This utility is responsible for translating text string to a desired language. After the translation, the mechanism also handles interpolation of data using the "*varname″orthesimpler*"varname" syntax. Common implementations of the domain make use of message catalogs, which provide the translations.

Reference: `zope.i18n.interfaces.ITranslationDomain`

See also: Utility, Domain, Message Catalog, Internationalization, Localization

Traversal

Traversal is the process of converting a path to the actual object given a base object (starting point). Traversal is a central concept in Zope 3, and its behavior can vary depending for which purposes it is used.

For example, if you traverse a browser URL, the traversal mechanism must be able to handle namespaces, views and other specialties and cannot be just a plain object lookup. It is also possible to change the traversal behavior of a given object by registering a custom traversal component for it.

Reference: `zope.app.traversing.interfaces.ITraverslAPI`
See also: Component, Namespace, View

Unit Tests

Unit Tests verify the correct functioning of the API and implementation details of a single component. Thereby the tests should not rely on any other component or a working environment. Unit Tests are the most commonly written tests and should exist for every component of the system. Every Zope 3 package should have a `tests` module that contains the unit tests.

Reference: `unittest`
See also: Tests, Doctests, Functional Tests

User

A user is any individual accessing the system via any communication method. While the user might be authenticated, s/he can be anonymous, as it is the case before the user logs into the system. The system associates various data with the user, including username/password, region, language and maybe even the computer from which s/he accesses Zope 3. Applications might associate other data with a user based on their needs.

Reference: `zope.app.pluggableauth.interfaces.IUserSchemafied`
See also: Security, Principal

Utility

This basic component provides functionality to the system that does not depend on state. If a particular utility is missing, it should not cause the system to fail.

Good examples of utilities are database connections, mailers, caches and language interpreters.

If you have troubles to decide whether you want to use a service or utilities for a particular functionality, think of the following: If the service is just a registry of components that will be used, then it is better to implement and register the components as utilities; the utility service functions like a registry as you can ask for all utilities that implement a certain interface.

See also: Component, Component Architecture, Service

View

Views are presentation components for other components. They always require a component (known as context) and a request to function. Views can take many forms based on the presentation type of the request. For browser views, for example, they usually just evaluate a template whose result (HTML) is returned to the client.

Reference: `zope.component.interfaces.IView`
See also: Component, Component Architecture, Presentation, Resource

Virtual Hosting

Virtual hosting, being HTTP-specific, allows one to run the Zope server behind another Web server while still handling all the target links of the request correctly. Virtual hosting is commonly used when Zope 3 is run behind an Apache installation, which might provide a different URL than the one Zope determines from a hostname lookup. The virtual URL can be specified via the "++vh++" namespace.

See also: Namespace

Volatile Objects

Volatile objects are object that appear in your traversal path, but are not persistent and are destroyed with the end of a transaction. They cannot be used to store persistent data. They are used as some sort of proxy, who looks up the data they represent as needed. The most obvious example is a SQL object that retrieves and stores its data in a relational database.

WebDAV

WebDAV is an extension to HTTP that defines additional HTTP verbs that are used to manage the available Web resources better. For example, it allows you to store any meta data about an object and lock/unlock it for editing. Zope 3 supports WebDAV partially.

Reference: `zope.app.dav.interfaces`

Widget

A widget is simply the view of a field, so it is specific to a presentation type. For the browser widgets are used to autogenerate input and display HTML code for the field of an object's attribute. Widgets can also convert the data coming from an external source (i.e. the browser) to a Python object that satisfies the field's type.

Reference: `zope.app.form.interfaces.IWidget`
See also: Field, Form, Schema, View

Workfbw

Workflows manage the state through which an object can pass and the processes that cause the state change. There are two approaches to workflows. The first one keeps track of the state of an object. The state can be changed to another using a transition. If a user has the required permission, s/he can cause a transition to another state. The other model (developed by WfMC) uses activities through which the object can go through. Only the first one has been implemented in Zope 3 so far.

Reference: `zope.app.interfaces.workflow`

ZConfig

This package allows one to write Apache-like configuration files that are automatically parsed into configuration objects. ZConfig is particularly useful for configuration that is likely to be edited by administrators, since they know this type of syntax well. Zope 3 uses ZConfig files to setup its servers, ZODB and loggers.

Reference: `ZOPE3/src/ZConfig/doc/zconfig.pdf`

Zope Configuration Markup Language (ZCML)

ZCML is a Zope-specific XML dialect used to configure and build up the Zope 3 component architecture during startup. All global components are registered through

ZCML, except for a few bootstrap components. ZCML can be easily extended by implementing new namespaces and directives.

Reference: `ZOPE3/doc/zcml`
See also: Component, Component Architecture

Zope Management Interface (ZMI)

ZMI is the formal name of the Zope 3 Web GUI that is used to manage content component instances and TTW software.

See also: Content, Through-the-Web Development

Zope Object Database (ZODB)

The ZODB stores all persistent data for Zope. It is a scalable, robust and well-established object database completely written in Python. Using the persistent mechanism, object data can be stored without any additional code. The ZODB has also two-phase transaction support and scalability is ensured through the distribution of the data to several machines using the Zope Enterprise Option (ZEO).

Reference: `ZODB`
See also: Persistent, Transaction

Zope Page Template (ZPT)

Page templates are the integration of TAL, TALES and METAL into Zope. You can write so called ZPT scripts that are executed at run time. ZPT's will provide a Zope-specific context, environment and base names for TAL, which makes them extremely useful. They are used as the primary tool to create HTML-based views in Zope 3.

Reference: `zope.pagetemplate.pagetemplate.PageTemplate`
See also: TAL, TALES

Appendix B

Credits

- **Stephan Richter** (srichter@cosmos.phy.tufts.edu) is the main author of the book.
- **Garrett Smith** (garrett@mojave-corp.com) provided the original version of the "Installing Zope Products" chapter.
- **Marius Gedminas** (mgedmin@codeworks.lt) provided much technical insight and example code for "Setting up a Virtual Hosting Environment" with Apache.
- **Ken Manheimer** (klm@zope.com) provided a different view on meta data in "Meta Data and the Dublin Core".
- **Brad Bollenbach** (brad@bbnet.ca) gave useful feedback and corrections to the chapter for "Internationalizing a Product" and other chapters.
- **Sutharsan "Xerophyte" Kathirgamu** (xerophyte@linuxnetworkcare.com) commented on early versions of the "Content Components" chapters and made suggestions.
- **Lalo Martin** (e-mail) corrected some typos.
- **Marcus Ertl** (e-mail) provided feedback to the early version of the Content Components chapters and corrected bugs.
- **Gintautas Miliauskas** (gintas@pov.lt) fixed numerous typos and corrected some mistakes.
- **Lex Berezhny** (LBerezhny@DevIS.com) proof-read many chapters and sent in corrections. He also provided general feedback about the book as a whole.
- **Eckart Hertzler** (hertzler.eckart@guj.de) updated and reviewed many of the content component chapters.

- **Paul Everitt** (paul@zope-europe.org) proof-read some chapters to point out unclear paragraphs and fixed grammer/spelling mistakes.
- **Max M** (maxm@mxm.dk) pointed out a mistake with a URL in the text.

Please let me know, if I forgot you!

Appendix C

Attribution-NoDerivs-NonCommercial License 1.0

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

   (a) "Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with a number of other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

   (b) "Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License.

   (c) "Licensor" means the individual or entity that offers the Work under the terms of this License.

(d) "Original Author" means the individual or entity who created the Work.

(e) "Work" means the copyrightable work of authorship offered under the terms of this License.

(f) "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

   (a) to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works; to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

   (b) The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

   (a) You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that alter or restrict the terms of this License or the recipients' exercise of the rights granted hereunder. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. You may not distribute, publicly display, publicly perform, or publicly digitally perform the Work with any technological

measures that control access or use of the Work in a manner inconsistent with the terms of this License Agreement. The above applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any reference to such Licensor or the Original Author, as requested.

(b) You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

(c) If you distribute, publicly display, publicly perform, or publicly digitally perform the Work or any Collective Works, You must keep intact all copyright notices for the Work and give the Original Author credit reasonable to the medium or means You are utilizing by conveying the name (or pseudonym if applicable) of the Original Author if supplied; the title of the Work if supplied. Such credit may be implemented in any reasonable manner; provided, however, that in the case of a Collective Work, at a minimum such credit will appear where any other comparable authorship credit appears and in a manner at least as prominent as such other comparable authorship credit.

5. Representations, Warranties and Disclaimer

(a) By offering the Work for public release under this License, Licensor represents and warrants that, to the best of Licensor's knowledge after reasonable inquiry:

i. Licensor has secured all rights in the Work necessary to grant the license rights hereunder and to permit the lawful exercise of the rights granted hereunder without You having any obligation to pay any royalties, compulsory license fees, residuals or any other payments;

ii. The Work does not infringe the copyright, trademark, publicity rights, common law rights or any other right of any third party or constitute defamation, invasion of privacy or other tortious injury to any third party.

(b) EXCEPT AS EXPRESSLY STATED IN THIS LICENSE OR OTHERWISE AGREED IN WRITING OR REQUIRED BY APPLICABLE LAW, THE WORK IS LICENSED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES REGARDING THE CONTENTS OR ACCURACY OF THE WORK.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, AND EXCEPT FOR DAMAGES ARISING FROM LIABILITY TO A THIRD PARTY RESULTING FROM BREACH OF THE WARRANTIES IN SECTION 5, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

   (a) This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collective Works from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

   (b) Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

   (a) Each time You distribute or publicly digitally perform the Work or a Collective Work, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

   (b) If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of

the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

(c) No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

(d) This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Appendix D

Zope Public License (ZPL) Version 2.1

A copyright notice accompanies this license document that identifies the copyright holders.

This license has been certified as open source. It has also been designated as GPL compatible by the Free Software Foundation (FSF).

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions in source code must retain the accompanying copyright notice, this list of conditions, and the following disclaimer.

2. Redistributions in binary form must reproduce the accompanying copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Names of the copyright holders must not be used to endorse or promote products derived from this software without prior written permission from the copyright holders.

4. The right to distribute this software or to use it for any purpose does not give you the right to use Servicemarks (sm) or Trademarks (tm) of the copyright holders. Use of them is covered by separate agreement with the copyright holders.

5. If any files are modified, you must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

Disclaimer

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Index