



Community Experience Distilled

Plone 3 Intranets

Design, build, and deploy a reliable, full-featured, and secure Plone-based enterprise intranet easily from scratch

Víctor Fernández de Alba

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

Plone 3 Intranets

Design, build, and deploy a reliable, full-featured, and secure Plone-based enterprise intranet easily from scratch

Víctor Fernández de Alba



BIRMINGHAM - MUMBAI

Plone 3 Intranets

Copyright © 2010 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either expressed or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2010

Production Reference: 1220710

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847199-08-9

www.packtpub.com

Cover Image by Faiz Fattohi (faizfattohi@gmail.com)

Credits

Author

Víctor Fernández de Alba

Editorial Team Leader

Mithun Sehgal

Reviewers

Ramon Navarro Bosch

Enzo Cesanelli

Leonardo J. Caballero G.

Matthew Wilkes

Project Team Leader

Priya Mukherji

Project Coordinator

Prasad Rai

Acquisition Editor

Rashmi Phadnis

Proofreader

Cathy Cumberlidge

Development Editor

Ved Prakash Jha

Graphics

Geetanjali Sawant

Technical Editors

Madhumita Singh

Arani Roy

Conrad Sardinha

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

Indexer

Rekha Nair

About the Author

Víctor Fernández de Alba has been an IT architect at UPCnet, the Barcelona Tech University (UPC) IT company since 2001. He has been leading the design and technical architecture of more than 100 projects undertaken by the UPCnet Plone Team since its inception in 2004. He has been teaching Plone to non-technical end users at the university for some time now. This allowed him to use this close experience with users for improved usability of web projects developed by its team. He also writes manuals about Plone targeted to end users.

The flagship project of UPCnet Plone Team is Genweb UPC. This project allows a fully featured, customized, and end-user ready intranets and public websites to be deployed in few short steps. It has helped to launch more than 200 Plone-powered sites at the university, including departmental web and intranets, university services, faculties, and other entities. In December 2009, the UPCnet Plone Team finished one of its most ambitious projects, and the main website of the Barcelona Tech University became a reality. It was developed focusing on scalability, high performance, and usability, using Plone.

Acknowledgement

I would like to thank my wife, Agata. This book wouldn't have been possible without her stubbornness, motivation, endless support, and unconditional love.

I want to specially thank my beloved mother, father, and brothers who have encouraged me throughout the years to better myself. I am who I am, thanks to them.

I would like to thank Javier Otero who has been a guiding light in my professional life, point of reference, endless source of knowledge, and above all, a friend. Thanks to Ramon Navarro who showed me the path to the Plone core and shared with me his dark magic. Thanks to both of them, who helped me review the book.

Finally, I want to remember those people at UPCnet who introduced Plone to me, and gave me the chance to work with it. This is a precious gift from the people who have, in one way or another, contributed to make this book a reality: the Plone community, the UPC Plone team, Vilabobo's team, the Packt Publishing team, specially to Rashmi, Ved, Prasad, and Madhumita, and my official book reviewers. Thanks a lot.

About the Reviewers

Ramon Navarro Bosch has been a computer science engineer since 2002 and is currently pursuing his PhD in Low Power embedded software. He has worked in the Universitat Politècnica de Catalunya (UPC) as System Administrator and Software Analyst from 1998 to 2006. During this period of time he has been involved in Samba, OLSR networks, C#-Mono, OpenWRT, Debian, LDAP, and Plone. He wrote a book about Mono development in C# using GTK for the Universitat Oberta de Catalunya. After working for a period of eighteen months for a Plone company called Headnet in Aarhus (Denmark), he returned to Spain with good knowledge and community contacts to start a Plone business. During the last three years he has been involved in some community projects (multilingual, bug solving, and local talks) and worked on some big projects for the UPC, such as the main page and the nearly 300 Plones infrastructure. He is also a teacher of software development at the UPC (using Python).

I thank my family for the support and for buying an Amstrad CPC when I was 10 years old.

Enzo Cesanelli has over nine years of experience in designing and testing web solutions. His humanist education, along with endless passion for new technologies, led him to focus on information architecture and user experience design.

Along his path, Enzo met Plone, which quickly became his engine of choice for most of his developed projects, namely, corporate websites, intranets, and web applications. A digital nomad, Enzo has just moved to London to broaden his personal and professional horizons, and to find new projects for Noiza, his communication agency based in Trieste, Italy.

T.S.U. Esp. **Leonardo J. Caballero G.** is a native from Maracaibo, Venezuela. He is a graduate of the "Academia de Software Libre" of Fundacite Mérida as "Especialista en Desarrollo en Software Libre" and also a graduate of the Colegio Universitario "Dr. Rafael Beloso Chacín" as "Técnico Superior Universitario en Informática". Currently he is a member of the CENDITEL Foundation community, where he serves as a Developer of Free Technology. He is an advisor to the Venezuelan government agencies regarding issues of community collaboration and free software development. He has experience in using Free Software since 2002. He has participated as a collaborator in the Internationalization process and Spanish localization in many Open Source and Free Software projects. He is an active contributor in Venezuelan projects, such as CANAIMA GNU/Linux, SAID, and others. Since 2006 he has been in charge of testing tools that facilitate the process of structuring and publishing content. He also contributes to the continuous improvement of source code in several third-party products of Plone CMS, OpenCore Software (used in the `CoActivate.org` website), and recently Django. He is a collaborator to PloneGov communities such as CommunesPlone (Belgium) and Open eGov (USA). He is a founding member of the Venezuelan Plone community. He has recently reviewed a book for Packt Publishing: *Plone Intranet*.

I want to thank God, all saints Ifa/Orisha, my family, Syra Lacruz, and Francisco Palm for their help and patience. I also want to thank CENDITEL foundation for learning and working with Plone, the members of all the communities including, "Pythonistas de Venezuela", "Plone Venezuela", "Plone ConoSur", and Plone for their comments, advice and patience.

Matthew Wilkes has been working with Plone since 2005, originally at Team Rubber, and as a freelance consultant under the name "Circular Triangle". During that time he was involved in every aspect of intranet site creation, from the initial planning stages to optimizing deployed sites to improve performance. He is also an active Plone community foundation member, serving the Plone 4 Framework Team and foundation membership committee, as well as performing documentation reviews and assisting the management of Plone.org itself.

*Per a tu Àgata,
la meua nena bonica.*

Table of Contents

Preface	1
Chapter 1: Introduction to Plone	7
What does Plone give me over other CMS solutions?	8
First surprise: not PHP, Python	12
Second surprise: not RDBMS, the mighty ZODB	13
The Plone community	14
Public websites, intranets, extranets, and the thin line between them	15
Summary	16
Chapter 2: Getting Started	17
Plone versions	18
Installing Plone	18
zc.buildout requirements	19
Quick start for the impatient	20
Plone unified installers	24
Windows	24
Linux	25
Mac OS X	27
Buildout	27
Distribute, setuptools, and eggs	28
PasteScript and ZopeSkel	29
Running buildout	30
Buildout directory structure	31
Setting up buildout.cfg	32
The buildout section	33
The zope2 section	34
The instance section	35
The zopepy section	36
Launching Zope	36
Summary	39

Chapter 3: Managing our Content	41
Plone visual layout structure	41
Header	43
Columns	44
Content	44
Footer	44
Anonymous versus logged in	45
Content management tabs	45
Content structure	46
Adding content	47
Standard Plone content types	49
Content metadata	50
Content settings	50
Managing content	51
Displaying views	52
Managing portlets	53
Summary	54
Chapter 4: Configuring our Site	55
Plone control panel	56
Mail control panel	57
Site	57
Users and groups	58
Security	59
Types	59
Add-on products	60
Content rules	60
Maintenance	61
Errors	62
HTML filtering	62
Language	63
Markup	63
Wiki formatting	64
Navigation	64
Search	65
Theme	65
Zope Management Interface	65
Control panel	66
Database management	67
Product management	68
Placeless translation service	68
Plone site—ZMI point of view	68

Installing new add-on products	71
As an egg via buildout	71
As a Zope 2 add-on product	75
Summary	75
Chapter 5: Managing Users, Groups, Roles, and Permissions	77
<hr/>	
One vision	77
Security entities	78
Roles	79
Global and local roles	80
Permissions	80
Global Zope user accounts	80
User self-registration	80
Managing users and groups	81
The user registration form	82
Managing users	84
Managing groups	85
Recovering user password	86
More control: managing ZMI	86
Administering users via ZMI	87
Administering groups via ZMI	88
Administering roles via ZMI	88
The sharing tab	89
Local role inheritance	90
Summary	91
Chapter 6: Managing Workflows	93
<hr/>	
Workflow entities	94
States	94
Transitions	94
Guards	95
Permissions	96
Assigning local roles to groups	96
Scripts	96
ZMI workflow management	96
Out-of-the-box workflows	98
Simple publication workflow	99
Community workflow	100
Community workflow for folders	100
One state workflow	101
Intranet workflow	102
Intranet workflows for folders	103

Workflow diving	103
States	104
Transitions	106
Variables	107
Worklists	108
Scripts	108
No workflow and multiple workflow use cases	108
Some useful workflow tools	109
DCWorkflowGraph	109
collective.wtf	110
collective.workflowed	111
Placeful workflow	113
Best practices	114
Make an initial blueprint first	114
Avoid developing on production servers	114
Start from an existing workflow copy	114
Use the tools shown for debugging	114
Test our workflow	114
Summary	115
Chapter 7: Securing our Intranet	117
Global or local roles?	118
Using global roles	119
Using local roles	119
Designing a sustainable role policy	119
A policy example	120
Restricting the use of the Manager role	120
Creating system administrator users for the Zope instance	121
Creating additional manager users of the Plone site	121
Granting other role permissions restricted to Managers	122
Local role delegation	122
Allowing non-managers to administer local roles	123
Choosing a workflow for our intranet	123
Restricting access to authenticated users	123
Building an example intranet workflow	124
Private state	125
Draft state	126
Intranet state	126
Transitions	127
Managing private content	129
Creating private sections	130
Workgroup areas	131

Third-party add-on products	131
Adding roles to the Plone UI	131
Using a custom product	132
Using collective.sharingroles	133
Summary	134
Chapter 8: Using Content Type Effectively	135
<hr/>	
Designing our intranet information architecture	136
Using collections	138
Creating a collection	139
Table of contents	141
Next/previous navigation	142
Presentation mode	142
Enabling the presentation mode	143
Formatting a slide	144
Third-party content types—best practices	145
A few golden rules	145
Ordering the "Add new" content type menu	146
Content type superseding	147
Maintaining usability	149
Upgrades	149
Summary	149
Chapter 9: Intranet Add-on Products	151
<hr/>	
Calendaring and extended events	152
Plone4ArtistsCalendar	152
Installation	152
Features	152
vs.event	154
Installation	154
Features	155
Form generators	157
PloneFormGen	157
Installation	157
Dependencies	158
How it works	158
Field types	159
Action adapters	160
Other content types in a form folder	160
Extensibility and third-party products for PFG	161
Captcha integration	161
Blogs	162
Quills	162
Installation	163
Features	163

Quills portlets	164
Configuring the blog	165
Scrawl	165
Installation	166
Features	166
Discussion board	166
PloneBoard	167
Installation	167
How it works	168
Adjusting permissions on Ploneboard for intranet use	168
Polls and surveys	170
PlonePopoll	170
Installation	170
How it works	170
Plone Survey	172
Installation	172
How it works	172
Document files management	174
ARFilePreview and AROfficeTransforms	174
Installation	174
Features	174
Additional software required	175
OpenXML	175
Installation	175
Dependencies	176
Summary	176
Chapter 10: Basic Product Development	177
<hr/>	
Building our own product	178
Naming our product	178
Creating the egg	179
Anatomy of a Plone product egg	180
Egg documentation files	180
Egg setup files	181
Main product content	182
ZCML configuration files	183
Making the product installable	183
The power of GenericSetup	185
Snapshots	186
Importing and exporting a particular product profile	188
Comparing snapshots and product profiles	188
Importing GenericSetup profiles from a product	189
Cloning content types via GenericSetup	190
Using a product to configure security	193

Defining role map assignment to permissions	193
Creating new workflows or modifying existing ones	194
Dexterity	196
Summary	197
Chapter 11: Content Rules, Syndication, and Advanced Features	199
Content rules	200
Adding a new rule	201
Assigning rules to folderish objects	204
Making any content type rule aware	204
Syndication	205
Enabling folder syndication	205
Accessing a secure RSS feed	206
Versioning	207
Changing versioning policy	208
WebDAV	209
Managing WebDAV access permissions	211
External editing	211
Installing the External Editor	212
Windows	212
Linux	212
MacOSX	213
Enabling external editing	213
Modifying helper software	213
Summary	214
Chapter 12: Theming our Intranet	215
Diving into Plone's page rendering	216
Acquisition from parents	217
Plone skins tool	218
Skins and layers	218
Acquisition in skin layers	221
Zope page templates	221
TAL	221
METAL	222
Viewlets	222
Managing viewlets	224
Composing a Plone page	225
Rendering the main_template.pt page template	225
Resource registries	227
CSS resource registry	227
JavaScript resource registry	228
Theming using third-party add-on products	228
GloWorm—add-on product for viewlet customization	228

Installation	228
Using GloWorm	229
CSSManager—add-on product for CSS and basic properties customization	230
Installation	230
Using CSSManager	231
CSS customization with base_properties sheet	232
Changing the logged-in tabs' attributes	232
Custom theme add-on products	233
Building our own theme add-on product	233
Installing the product	234
Customizing Plone skin layer resources	236
Enabling CSS debug mode	236
Customizing the site logo	236
Customizing the logo image and adding a new one	236
Customizing the plone.logo viewlet	237
Customizing Plone CSS	238
Resetting Plone CSS	239
More about customizing viewlets	240
Using Generic Setup to customize a theme	241
Theming—best practices	241
Summary	242
Chapter 13: Deploying our Intranet	243
<hr/>	
Deployment buildouts	244
Buildout base configuration	245
Adding a versions file	245
Caching extended configuration	246
Using the newest directive	246
Adding ports and hosts names sections	247
Adding process owners section	247
Changing the ownership of buildout folder	248
Common administration tasks	248
Backing up and restoring database	249
Database packing	250
Rotating the log files	250
Scheduling	252
Virtual hosting	252
VirtualHostMonster	252
Virtual hosting a root domain	253
Virtual hosting a domain subdirectory	254
Small intranet deployments	255
Monolithic Zope	255
Performance	256

Scalability	256
Buildout for small deployments	256
Small deployments layer diagram	256
Medium intranet deployments	257
ZEO (Zope Enterprise Objects)	257
Adding a ZEO server to our buildout	257
ZEO clients	258
Scalability	258
Performance	258
Adding ZEO clients to our buildout	258
Load balancer	260
Supervisor to rule them all	263
Using Supervisor	264
Modifying the web server settings	265
Medium deployments layer diagram	265
Large intranet deployments	265
Adding cache to our deployment	266
Products.CacheSetup add-on product	266
Cache server	268
Building and configuring Varnish	269
Default VCL configuration template file	270
Modifying the web server settings	272
Spanning services in separate servers	273
Increasing the ZEO client instances	273
Updating balancer configuration	274
Setting LDAP as an external user database	274
Large deployments service layer diagram	277
Summary	278
Index	279

Preface

Plone is a highly extensible Content Management System built on Zope application server, which is written in Python. Plone is very suitable for building intranets. No matter what size, or purpose, it offers a solution to the most common intranet needs, and more. Although it shows its real power in medium and large-scale corporate intranets, we can take advantage of Plone even in small-scale scenarios, such as small work groups, software projects, or research teams.

If you've never built an intranet before, you don't even have programming skills, or you don't have any experience with CMS, don't worry! This is the most suitable book for you.

This book will give you a complete overview of how to build and design your intranet, focusing on security and usability. It will guide you through the initial setup, Plone basics, security, and workflow-related topics, and ends with the most common deployment techniques.

Learn how to make effective use of content type for your intranet. Know how to manage the life cycle of your documents and content in general, using workflows. Manage security and access your content granularly. Learn how to choose the right add-on products for your site, and how to use it in your intranet efficiently. And at the end, know how to deploy your intranet and make your site live.

This book is targeted at people with no previous experience in Plone. There is no need for any programming experience or CMS knowledge.

What this book covers

Chapter 1, Introduction to Plone, introduces readers to Plone, what it is, for what it's used, its main features, and why it is one of the best CMSs to build an intranet.

Chapter 2, Getting Started, teaches us how to download the software requisites, install them, and run our own instance of Plone, no matter which software platform we may have.

Chapter 3, Managing our Content, talks about the basics of Plone. Before we advance to building our Plone intranet site, we should know how to manage content, as well as the basics of Plone. This section only outlines the basics of Plone and does not go in depth. However, it gives us sufficient information to have a broad knowledge of Plone.

Chapter 4, Configuring our Site, covers the most important topics about the advanced configuration of our Plone intranet and other advanced topics. It shows us how to manage our Plone site. We will learn to be confident in managing our site using the Zope Management Interface (ZMI).

Chapter 5, Managing Users, Groups, Roles, and Permissions, discusses security, and how to deal with it, which is one of the most important topics when building an intranet. In this chapter, readers will learn how security works in Plone, how to manage local users (CRUD operations), and the mechanisms that Plone makes available to users in order to manage their data, such as profile data and passwords. We will also learn about groups and roles, and the basics of Plone security.

Chapter 6, Managing Workflows, is a detailed chapter dedicated exclusively to workflows, best practices, and how to manage, create, and modify them.

Chapter 7, Securing our Intranet, covers topics based on the experience earned whilst working with intranet users and common basic intranet needs.

Chapter 8, Using Content Type Effectively, covers common intranet use cases and shows us how to deal with them in Plone. We have to be very careful about the content type that we make available to our users. Failing to do so will lead to really difficult migrations, product conflicts, and user confusion.

Chapter 9, Intranet Add-on Products, covers a suitable stack of products addressed to intranets. It shows the most reliable and fully-featured set of Plone's third-party add-on products. It covers internal blogs, group discussions, wikis, and knowledge bases form generators, surveys and polls, creation of shared calendars, document file helper applications, and so on.

Chapter 10, Basic Product Development, is a brief introduction to product development. It does not give an in-depth knowledge, but it aims to introduce the reader to the basics, and to the right resources to learn more about the topic.

Chapter 11, Content Rules, Syndication, and Advanced Features, talks about advanced content features such WebDAV, access to bulk upload and download content, content rules, and syndication, amongst others.

Chapter 12, Theming our Intranet, lists the best practices for theming an intranet, focusing on performance. However, an in-depth chapter about theming is out of the scope of this book. It also talks about basic information with reference to resources where the reader will learn more about this subject.

Chapter 13, Deploying our Intranet, covers how to deploy our intranet, based on its capacity, its needs, the number of intranet users, and its size.

What you need for this book

You will need a Python-enabled environment, which is very easy to set up under Linux and Mac OS X. Windows users should make sure that they install all the prerequisites in order to have a fully functional Python command-line interpreter, such as Linux and Mac OS X users have.

All code and examples shown should run on any platform. You will need internet access to download all the software and dependencies required to run Plone.

All prerequisites are described in depth in *Chapter 2, Getting Started*.

Who this book is for

This book is for anyone who needs to build an intranet with no limits on capabilities or features. Even if you don't have previous CMS experience or programming skills, this book is for you. Targeted at beginners with no previous experience with Plone, this book will teach you step-by-step, and at the end you should have a full-featured, reliable, and secure intranet.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "The name of this database is `Data.fs.pack`."

A block of code is set as follows:

```
[zeoserver]
recipe = plone.recipe.zope2zeoserver
zope2-location = ${zope2:location}
zeo-var = ${buildout:directory}/var
zeo-address = ${ports:zeo-server}
```



When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:



```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  xmlns:tal="http://xml.zope.org/namespaces/tal"
  xmlns:metal="http://xml.zope.org/namespaces/metal"
  xmlns:i18n="http://xml.zope.org/namespaces/i18n"
  lang="en"
  metal:use-macro="here/main_template/macros/master"
  i18n:domain="plone">
```

Any command-line input or output is written as follows:

```
$ paster create -t plone3_buildout deployment.buildout
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "In the **Groups** tab we can find a similar functionality as the **User** tab."

 Warnings or important notes appear in a box like this. 

 Tips and tricks appear like this. 

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.


To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail suggest@packtpub.com.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

[	<p>Downloading the example code for this book</p> <p>You can download the example code files for all Packt books you have purchased from your account at http://www.PacktPub.com. If you purchased this book elsewhere, you can visit http://www.PacktPub.com/support and register to have the files e-mailed directly to you.</p>]
---	---	---	---

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Introduction to Plone

Plone rocks!

If we have this book in our hands, we probably already know it, or have heard about its excellence. First things first, we can't write a book about Plone without introducing it to you properly. Plone has become a huge phenomenon in the last few years. I would like to show you its history, some facts, and who is who in the Plone world.

Plone is a mature open source **Web Content Management System (WCM, WCMS, or Web CMS)**. Alexander Limi, Alan Runyan, and Vidar Andersen began the Plone project in 1999. It was conceived as a usability layer on top of the **Zope Content Management Framework (CMF)**. It has been released under the **GNU General Public License (GPL)**, and today is one of the most active open source projects in the world driven by more than two thousand developers and collaborators. In order to ensure that Plone will always remain as an open source project, the Plone Foundation was created in 2004. As a non-profit organization, it exists to protect Plone's intellectual property and trademarks, and is in charge of taking the leadership, making all of the important decisions about its design, features, and philosophy.

Nowadays, a lot of organizations, enterprises, NGOs, universities, and many others have trusted in Plone to host their websites or intranets. To mention some of them: NASA Science, Novell, Nokia, Amnesty International, the Brazilian government, the Nordic Council, and so on.

Plone is built on usability; thus, the learning curve needed to use it is very low steeped because of its unique design at various levels. It makes the user experience very pleasant for a non-technical user, making web publishing an easy process. This goal is achieved, thanks to a well-designed user interface and the site building process based on a folder tree hierarchy.

Plone is built on top of Zope and benefits from all its features. Zope is written primarily in the Python programming language and had the honor to be the first open source web application server. One of Zope's most important features is the **Zope Object Database (ZODB)**, its object oriented database. This and many other features make Plone a highly extensible, scalable, multiplatform Content Management System.

Plone is really easy to learn, use, install, and extend. It comes with simplified installers for Windows, Mac OS, and Linux, along with other operating systems. There are more than 1000 add-ons created by the Plone community available on the web, extending Plone in every way imaginable, from authentication plugins to collaborative tools, through (almost) every functionality or feature we may need for our website. It is available in over 40 languages and provides a multilingual content engine (via another add-on) in order to manage the translations of our site.

Plone is really suitable for building intranets. No matter how large or for what purpose, it offers solutions to the most common intranet needs, and more. Due to its object orientation, we can define fine-grained permissions to users or groups, and build complex security structures inside our intranet. On top of that, it provides us with an easy user interface to manage security and permissions. At the same time, due to its large amount of add-ons our intranet will be able to provide the most popular collaboration tools and the most useful productivity tools available.

Additionally, Plone is based on technology standards, such as XHTML, CSS, or Dublin Core. It integrates well with other standards such as LDAP, Web Services, SQL, Active Directory, and so on. In the accessibility aspect, it meets or exceeds US Government 508 and **World Wide Web Consortium's (W3C's)** WAI-AA standards.

With regards to security, Zope and Plone have a technological edge that has helped it attain the best security track record of any major CMS (source: CVE <http://cve.mitre.org>).

What does Plone give me over other CMS solutions?

As we already know, CMS stands for Content Management System, but this definition is very broad and is applied to a large set of solutions. Many authors tend to split them in several categories, the most important are: **Web or Portal Management Systems (WMS)** and **Enterprise Management Systems (EMS)**.

Drupal, Joomla, Plone, dotNetNuke, to mention a few of them are WMS. There are others targeted to specific use cases of content production, such as Wordpress or Zine, which are aimed exclusively at blog publishing. Although it is common to call them WMS, there are some authors that treat them as a special case.

WMS software is a web application for creating and managing large, dynamic collections of web material (HTML documents and their associated resources). Usually they provide authoring (and other) tools and rich user interfaces designed to allow users with little or no knowledge of programming or markup languages to create and manage content with relative ease of use. A database is used to store content, metadata, and in some cases other types of artifacts as code or templates. Usually, a presentation layer displays the content based on a set of templates.

One of the most distinctive features of the WMS market is the way the users manage content. In fact, there are two approaches:

- Backend
- In-place editing

In the backend approach, the system makes available to the user a special interface for managing the content resources, as well as administering the system. In the in-place editing, the user can manage the object directly through the frontend user interface, using special widgets the system provides for that purpose.

Another important feature is how we store and choose to show content on our site. Most CMS rely on taxonomy, tagging each document with an ID, which is used to specify where we want to position it on our site. This is the approach chosen by Drupal, Joomla, and so on. They also use the concept of **asset library**, a place to put resources for a later use on our site.

There are a lot of WMS that are backend driven, and although some of them allow some kind of in-place editing, the experience is not always as pleasant as we would desire. Some of them rely on taxonomy to build the site structure. **Taxonomy** is the practice of classifying objects according to natural relationships. This method fits into the brains of technical users, but it's not easy to understand for non-technical users due to the complexity of the concept and, often, due to a poor and non-usable interface.

We can find these features and more in any modern WMS. However, what are the differences between WMS and EMS? The boundaries between WMS and EMS are becoming more blurred in recent times. Enterprise Management Systems are targeted to capture, manage, store, preserve, and deliver large amounts of documents, and treat them as individual entities. Often these documents are stored in XML, which provides easy integration and interoperability with other systems. They are focused on the management of the life cycle of a document and they often integrate powerful business process management (BPM) tools. They can also provide archiving and Public Key Infrastructure integration. To name a few: EMC | Documentum, FileNet, or OpenText.

Although we can find some of these features in a WMS, or we could just implement them, we have to keep our needs in mind. We have to decide what kind of tool is the most suitable for our requirements and meets our needs. Not all needs end with the choice of a general WMS, for example, if we need a simple website with the brochure of our business, maybe we will choose a WMS like Joomla or Drupal. If our organization is related to the government or it's a big enterprise where we need integrated document management, with the highest audit and control requirements, and we are not interested in publishing our site, then we probably need an EMS.

From a broad point of view, an intranet is an access-controlled site along with optional external publishing capabilities, which has the following key features:

- Usability
- Security
- Collaboration tools
- Productivity
- Content life cycle

Plone excels in all of them. Let me show you how.

For non-technical users, it is essential to provide the simplest and a highly usable user interface, relying on concepts that they already know and they will be able to easily extrapolate to our use case. Here is where in-place editing and a tree-based hierarchy come into play.

A tree-based hierarchy is a repeating concept every user has to learn when dealing with computers because of the structure of file systems. It is easy for users to apply this concept when it comes to managing content. Plone uses a tree-based hierarchy for organizing its content and transparently storing it as objects, provided by the ZODB, the Zope's object oriented database. From the user's point of view every Plone site has a root folder, which is a real physical place where they could place content, and eventually folders that would be physical containers of more content. First-level folders would become the website's sections. In my experience, users are very confident in that scenario, and they learn the concept almost immediately. They can even do the same basic operations over content such as cut, copy, and paste as they used to do with files in a file system environment.

Another key aspect people can expect from an intranet is security. I'm not talking about restricting access to our site to authenticated users only. It's about being able to define authorization over content objects by specifying a complex permission structure on them too. Plone provides a granular and highly configurable security engine. Permissions are defined at the object level, assigning specific permissions to every single object. The **Plone pluggable authentication service (Plone PAS)** takes care of the connection to the most common authentication methods (Local, Active Directory, NTLM, LDAP, SQL, Novell, and so on) and its pluggable architecture can be extended to support others.

Enabling workgroups and collaboration is a must in an intranet. We have a constant need to share our daily work and we often need a centralized place to do so. A few years ago, the initial purpose of intranets was to be mere information containers. Today, we are constantly asked to improve their capabilities and we are continuously making them more powerful. Forums, blogs, form and survey generators, contact management, resource bookings, travel expenses; all of these could be nice features to add to our intranet. Intranets must provide us with a way to make our daily work more easy and reusable. Plone has a long list of add-ons that can help us in this matter. In short, probably the most used topic in IT in the beginning of the 21st century – Web 2.0!

Productivity is not a joke for any enterprise or organization. We are living in a world that demands the highest standards. We must work efficiently and achieve the highest quality results by spending the least amount of money possible. Having the right tools to accomplish this is not an option, but a must. Again, there are a lot of add-on products that provide these kind of tools. Many of the applications that we mentioned earlier in this chapter could help us achieve that.

A very useful feature that is able to define the life cycle of content is called workflow. Workflow is composed of **states** and the **transitions** between them. The states define a set of permissions and visibility of the content, and the transitions define the actions to be taken when content's state change is triggered. Plone provides a highly customizable workflow engine, and the most common ready-to-use OOTB workflows. The right use of workflows is the key to a successful intranet.

First surprise: not PHP, Python

As we have seen, nowadays, there are a lot of successful WMS applications and the most popular ones are PHP based. Joomla and Drupal are good examples; we could even mention some extremely successful websites such as Facebook, which are PHP based. The popularity of PHP is not a surprise; good documentation, a shallow learning curve, its similarities in terms of keywords and language syntax with other popular languages such as C, its web oriented architecture, and a long career in the IT world are its credentials.

The first thing we notice about Plone is that it is, in fact, a Zope application. To be more exact, it's a Zope product. And because of that, against all odds, it's Python based.

I'm a Python lover, and this romance has its reasons. Having reached this point, I could just begin another little war between PHP and Python, but I won't. There are a lot of discussion boards on the Internet about this subject, and we can find them easily. Instead of opening Pandora's box, I will talk about the excellences of Python, and why it excels in doing its job.

Python is a dynamic object-oriented programming language that can be used for all kinds of software development. Guido van Rossum, former employee of Zope Corporation and now working for Google, created it in the early 1990s at Stichting Mathematisch Centrum in the Netherlands. As we may already know, Google has made a strong bet on Python, and is using it in Google Apps Engine and presumably in the core of its most popular services.

It offers integration with other softwares and tools, and comes with extensive standard libraries. It features a very intuitive and easy syntax and can be learned in few days. Following are some of its key features:

- Very clear, readable syntax
- Strong introspection capabilities
- Intuitive object orientation
- Natural expression of procedural code

- Full modularity, supporting hierarchical packages
- Exception-based error handling
- Very high level dynamic data types
- Extensive standard libraries and third party modules for virtually every task
- Extensions and modules easily written in C, C++ (or Java for Jython, or .NET languages for IronPython)
- Embeddable within applications as a scripting interface



Having reached this point, don't panic! There is no need to have a good level of Python knowledge to follow this book, as we will not enter deeply into development matters. If you are a proficient PHP or other language programmer, then welcome! Don't feel overwhelmed about having to learn a new language. As we said, Python is easy to learn and has all the standard features we would expect from a modern programming language.

Second surprise: not RDBMS, the mighty ZODB

Yet another surprise? Yep, I will tell you just one thing: you can forget all you know about relational databases, SQL query languages, tables, fields, and stuff like that. Let me introduce you to the mighty ZODB! If you don't know anything about it, don't worry.

The ZODB is an object-oriented database for storing Python objects transparently and persistently. It is included as part of Zope, but can also be used independently out of Zope. The reason for not using a **relational database management system (RDBMS)** is easy to understand. It is more natural for a Content Management System to store data in objects than to rely on an abstraction layer that converts the document object we are storing to fields in a table (or fields across several tables), and again when we retrieve the data from the object. In all aspects, it is easier to store the object directly and transparently in the database as an object.

Plone stores the site content, components, templates, and code needed in the ZODB. The content is saved in the database following a tree-based hierarchal structure from the root of the Plone site. Every item of content is an object, and the associated metadata (title, description, body, and so on) are its attributes. As we said before, for some applications like CMS, it is more efficient to store content in this more natural way, as the type of entities we are managing is most of the time structured hierarchically. ZODB users don't have to know where and how all this is stored, because the entire persistence layer is transparent to them.

Following are its main features:

- Transactions
- History/undo
- Transparently pluggable storage
- Built-in caching
- Multiversion concurrency control (MVCC)
- Scalability across a network (using ZEO)

On Zope, every transaction is stored and is not written in the database until it is committed. This means that if an operation fails while it is running, the database remains unchanged, as no data is committed to the database until the end of the operation.

As every transaction is stored, we can see the history of all operations for a container. We can undo the transaction and revert to the state before that transaction was made. So, if something goes wrong, or if someone deletes a folder by mistake we can simply recover it by undoing the transaction.

Believe me, these two features have protected us from a lot of nasty headaches.

The Plone community

Now it's time to introduce you to the Plone community. You will find that it is an exquisite cocktail and its recipe follows like this: lot of highly-talented people immersed in an exciting and continuously challenging project, along with some crazy geeks and high level third-party add-ons contributors, two spoons of wisdom provided by the Plone Foundation, and finally shaken, not stirred, with a high amount of friendship.

This community has the key to all our questions about Plone. There are a lot of people eager to help those who ask for it. We can reach them via mailing lists or chat room (visit <http://plone.org/support>). It is also the entry point to discuss every aspect of Plone, its design, features, and future. You may take a look at what is said in the planet of blogs (<http://planet.plone.org>), what is being planned for future releases of Plone, what is happening in the community (<http://plone.org/news> and <http://plone.org/events>), and you can be a part of it.

There are lots of ways to support and promote the community; one of them is attending periodical events such as conference, symposia, and sprints. There are many others, such as organizing local meetings about Plone; just a little digging is needed to find out if there's one in our city, and if there is none, then create one!

The Plone Conference is the most important event inside the Plone world. It is held every year since 2003, and it is organized by the local community of a designated city. Attending a Plone Conference is the best chance one has to meet the Plone community in pure state. Having firsthand experience in how it is designed, how it is made, how it works, and the possibility to take part in the development of new Plone features is invaluable.

One of the key points behind Plone being one of the most active open source projects is not any accident—it's the Plone community without any doubt.

Public websites, intranets, extranets, and the thin line between them

There was a time when public websites, extranets, and intranets were separated by well-known and defined borders. These borders were defined by file system permissions set at the OS level, and were statically defined and not accessible by users.

Nowadays, the difference between them in most cases is a thin line, often defined by object properties (such as a state or user security permissions) that determines the visibility and the access rights for each object of our site. As we said earlier, Plone allows us to set up a property called "state" at object level. This state will have a set of permissions associated with it that will define the access to the object itself. These permissions are part of a bigger set of security-related properties that allows us to manage object security more accurately and helps us to have high granular security management.

As we can determine the visibility of each object, we can have public access objects living in the same folder, along with others with restricted access rights.

This leads us to the paradox of having a public site that could easily have an intranet section protected with access rights, or a big intranet where it is permitted to publish certain content or expose public sections to anonymous users.

Changing states of content in the Plone UI is only two clicks away. A state is the main entity of a workflow and, among other things, it defines the user permissions of the object. Each content type can participate in a workflow (or none if we don't want to) and Plone provides us with a way to define and manage them. Plone is shipped out-of-the-box with some common use workflows, but we can define and manage them to fit our needs.

Summary

In this chapter, we have learned all the basic concepts of Plone, along with its features, a few of which were actually surprising for newcomers. We also introduced the Plone community and why Plone is a good option to build out intranet using Plone.

In the next chapter, we will cover the installation and first steps with Plone.

2

Getting Started

Now, let's get our hands dirty. In this chapter, we will learn how to install and run Plone. We will also learn what software prerequisites are needed and which are the most useful tools. Our main goal will be to get acquainted with the correct way to install Plone and feel comfortable with it. We will also introduce the **Zope Management Interface** as the main Zope administration tool.

This chapter presents the following topics:

- Software prerequisites and their versions
- Quick start
- Unified installers
- Using buildout
- Running our first instance of Plone

From now on, all examples and exercises are intended for implementing on a development server or on a test computer. Although the settings shown in the following chapter could be used on production servers, it is recommended to follow strict guidelines, procedures, and consider good practices when installing a production server. In *Chapter 13, Deploying our Intranet*, we will learn all the insights on how to set up a production server.

Keep in mind that Plone, Zope, and Python are multiplatform software, so we have no special limitation as to hardware or operating system. Today Python is available for any major operating system, and by extension so are Zope and Plone.



From now on in the examples and console code shown in this book we will show a Bash interpreter shell, though we will cover Windows syntax when it differs significantly. Bear in mind that path separators on Windows are backslashes (\) and the command prompt will be a > sign, while other environments use forward slashes (/) and command prompts will be a \$ sign.

Plone versions

At the time of writing this book, Plone's latest published version was 3.3. Throughout the book this version will be used as a reference, regardless of the minor revisions that might be published in the future, which are not expected to represent any major changes in functionality or interface.

Plone 4 is scheduled to debut at some point in summer 2010 and it will include some very interesting features as follows:

- Global better performance (due to use of Python 2.6 and Zope 2.12)
- Added TinyMCE for editing
- Use blobs file type as part of the core
- Unify folder implementations
- And, many other enhancements

Plone 4.0 is a new major feature release, which builds on the Plone 3 release series. Its focus is on updates on the base technologies, better support for large data volumes, and a variety of end-user features. In most cases, all the concepts covered in this book are valid for both versions 3 and 4.

Installing Plone

Basically, in order to install Plone we need a full installation of Python, a Zope server instance, and of course, all Plone's module dependencies.

It is worth noting that each version of Plone requires us to install a specific major Python version on our machine (minor revisions are usually safe). For example, Plone 3 requires version 2.4 of Python, whereas Plone 4 needs 2.6. For now, Plone is not expected to use the recently released Python 3.0.

As to what relates to Zope, we'll need at least version 2.10.6 for Plone 3, whereas Plone 4 requires Zope 2.12.1.

The following table summarizes the versions needed for each version of Plone:

Plone version	Zope version	Python version
Plone 3	Zope 2.10	Python 2.4
Plone 4	Zope 2.12	Python 2.6

We can install Python and Zope and set up a Zope instance, and then install Plone and its dependencies in the Zope instance. Doing it by hand is complex and tedious. However, there are easier and more efficient ways to install Plone, and we will discuss them shortly; all of them provide us with some interesting advantages. We will focus specially on one of them, `zc.buildout`, as it proves to be the more useful one.

zc.buildout requirements

In order to use `zc.buildout` to install Plone we must have the following tools and libraries at hand:

- A complete Python installation, with the development libraries installed: A full Windows installation will suffice and in a Debian/Ubuntu Linux the `-dev` package of the corresponding Python will suffice (for example, `python2.4-dev`).
- A C compiler: If we are in a windows platform, we may want to install `mingw32` (<http://www.mingw.org>) and the Python Win32 extensions (http://downloads.sourceforge.net/pywin32/pywin32-210.win32-py2.4.exe?modtime=1159009237&big_mirror=0). For Linux/Unix platforms we may probably have it already installed, otherwise just refer to your OS software installation system. In Mac OS X we must install Xcode; we can either find it in our installation Mac OS X DVD or download it from the Apple website.
- `Distribute`: An extension of "`distutils`", Python's built-in packaging system. More information on the best way to install it is available in the next section.
- `ElementTree`: An XML processing library. Most operating systems have packages for this library but it can also be downloaded from <http://effbot.org/zone/element-index.htm> or installed through `easy_install` `distribute` script.
- The Python Imaging Library (<http://www.pythonware.com/products/pil>): This must be installed for the Python version we are going to use.
- A command line or interpreter: Like Bash in Linux/Unix environments or `cmd` in Windows boxes. Remember to set up the `PATH` environment variable of our operating system to point to the Python and C compiler binaries to access them from any path in our system.


Quick start for the impatient

If we have already checked out all the requirements, let's make a quick setup. We will find that setting up a Plone instance is a matter of minutes.

Open our command interpreter, and type the following commands:

```
$ wget http://python-distribute.org/distribute_setup.py
$ sudo python2.4 distribute_setup.py
```

These commands will install `setuptools` in our system along with its support script `easy_install`. They provide the functionality to install any Python module packaged in `.egg` format in our system. We will install `setuptools` by downloading a script (`distribute_setup.py`) that will install it. To do this we can use `wget`, as shown in the example, or we can download it using our favorite web browser. Then we must execute it with the correct Python version: as we are installing Plone 3.3 we must use Python 2.4.

 Maybe we have already installed the `setuptools` package, especially when it comes to any Linux distribution. In this case, it is recommended to *upgrade* to the latest `setuptools` version by running the command lines exposed, as it's common that it will have an outdated version of the package.

In the next step, we will install the `elementtree` module using `easy_install` script (if we have not installed it before) and `ZopeSkel`.

```
$ sudo easy_install-2.4 elementtree
$ sudo easy_install-2.4 ZopeSkel
$ paster create -t plone3_buildout myplonebuildout
```

It is a collection of skeleton templates for the **PasteScript**, a very powerful Python developer tool. Amongst other features, PasteScript can generate skeletons of source files and folders for our software applications. They are configured via a wizard. We will use PasteScript's script `paster` to create a skeleton for our `buildout` instance of Plone. The `create` command consists of the name of the template (`plone3_buildout`) and the name of the `buildout` directory we are creating. The script will ask us a few questions about our resultant `buildout` instance:

```
Enter plone_version (Which Plone version to install) ['3.3.4']:
Enter zope2_install (Path to Zope 2 installation; leave blank to fetch
one) ['']:
Enter plone_products_install (Path to directory containing Plone
products; leave blank to fetch one) ['']:
Enter zope_user (Zope root admin user) ['admin']:
```

```
Enter zope_password (Zope root admin password) ['']: admin
Enter http_port (HTTP port) [8080]:
Enter debug_mode (Should debug mode be "on" or "off"?) ['off']:
Enter verbose_security (Should verbose security be "on" or "off"?)
['off']:
Creating template plone3_buildout
[...]
```

We will only have to inform the desired Plone's version we want to install. It is advisable to set the latest stable version available (in the previous example, we are setting up 3.3.4). In the case of Zope root admin user's password, the convention is to use `admin`. We will answer the remaining questions with the default option.

```
$ cd myplonebuildout
$ python2.4 bootstrap.py
```

Once we have generated our buildout structure, we will move to our recently created directory `buildout`, and run the script `bootstrap.py` with the proper Python version. This will generate some additional directories and installation specific scripts that we will use later. After that, we must run the `buildout` script:

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

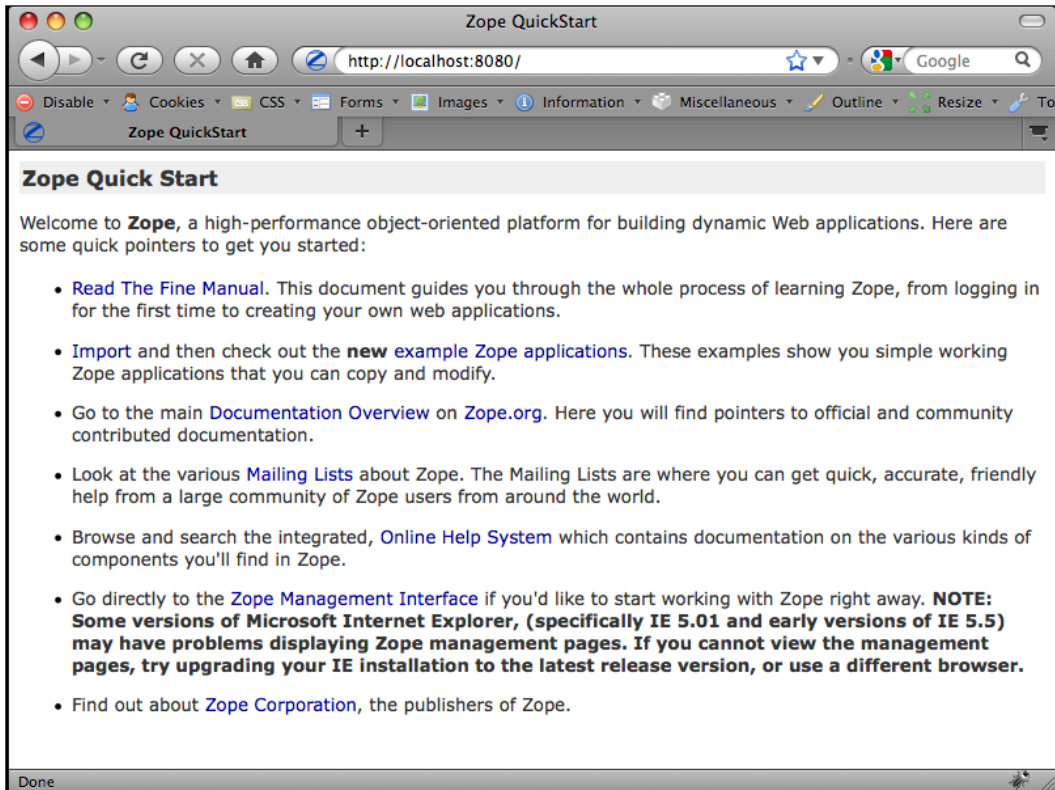
Once we are done, we can launch the Zope server process with the next command line:

```
$ ./bin/instance fg
```

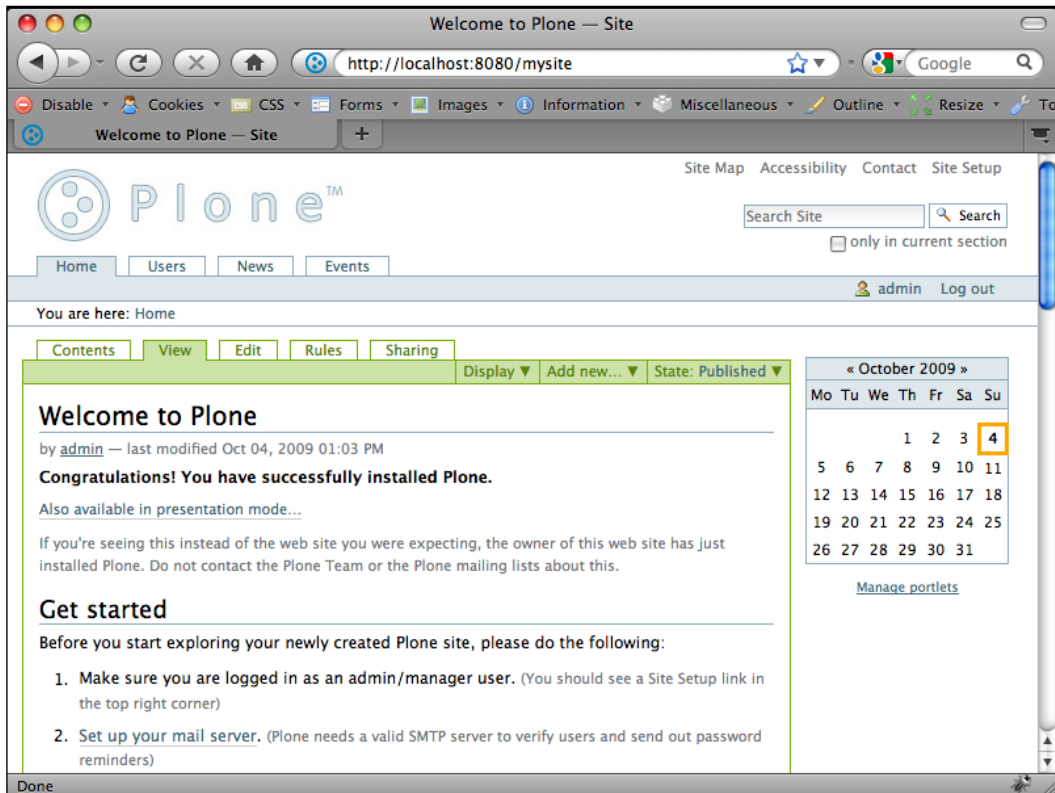
We will wait until we see the following message:

```
INFO Zope Ready to handle requests
```

Then open any web browser and point to the URL: `http://localhost:8080`. We should see Zope's splash screen by default, which looks like the following screenshot:



We will reach the **Zope Management Interface (ZMI)** through the following URL: `http://localhost:8080/manage`. The browser will ask us to enter the username and password that we have previously configured during the buildout creation (for example, the user can be `admin` and password also `admin`) to access it. Once logged in, within the ZMI, select **Plone Site** from the **Add new...** drop-down menu near the top right to add a Plone site. Once finished, if we were to call it *mysite*, it would be accessible from `http://localhost:8080/mysite`.



Congratulations! We have just created our first Plone site! It's a good moment to take some time to familiarize ourselves with the surroundings, and have our first hand experience with Plone.

We have just learnt (the express way) how to install Plone via `zc.buildout`. We will cover two ways of installing Plone, via `zc.buildout` and via **Plone's Unified Installer (PUI)**. Plone 3.3 Unified Installer is based on buildout, hence, when we use PUI we will be using `zc.buildout`. PUI incorporates some automatic procedures that are useful for beginners. We can start with PUI, but as we gain experience, sometimes it's more flexible and efficient to use a standard Plone buildout as we have seen a moment ago.

Plone unified installers

Along with each published Plone version, a Unified Installer is released for the most popular operating systems. In fact, it is a kit that will help us install Plone easily, featuring a small installation wizard. This wizard installs all the prerequisites needed for Zope and Plone to work properly, and a set of scripts and programs for assistance to start and stop services related to the application. We can find it at <http://plone.org/products/plone>.

The unified installer is ideal for beginners to Plone, to take their first step and begin learning how it works. Regardless of the installation platform we choose, it's very simple to use and requires virtually no knowledge or no additional configuration.

We will show how PUI works in Windows, Linux, and MACOS X, as each one contains a number of features worth mentioning. Because of the fact that they are all based on `zc.buildout`, we will not discuss any advanced aspect about how buildout works until the next section, when it will be discussed in depth.

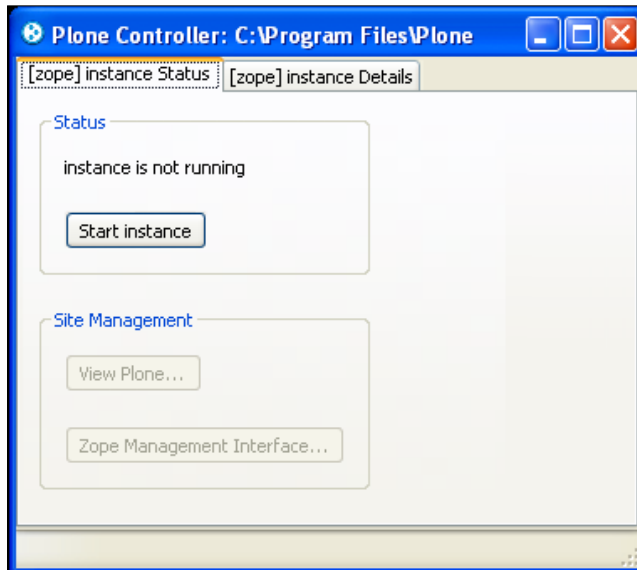
Windows

Once downloaded, run the installer `Plone-3.3.x.exe`. The setup wizard will ask us only two things:

- The location of files on the local machine (by default, on `C:\Program Files\Plone`)
- The username and password of the Zope administrator user account

The program will install a preconfigured version of Python, a precompiled Zope server, and all the modules required to run Plone and its dependencies. The wizard will locate the files in the chosen directory.

It installs a service on the machine that starts the Zope server when we boot the machine. We can control the state of the server from a small application called `Plone Controller`, which is installed in the Windows programs menu.



This application has shortcuts to access directly our Plone Site and the **Zope Management Interface**.

Linux

Download the installer package file in our local machine. It's packaged in a `.tgz` file, so we will have to decompress it in our desired location:

```
$ tar xvfz Plone-3.3.5-UnifiedInstaller.tgz
```

This kit includes the source code of both Plone and its dependencies. Unlike the Windows installer that bundles precompiled versions, this kit will compile from source Python, Zope, and Plone. The Python installation is done in such a way that it does not interfere with any Python installation that is already installed in the machine.

The kit also includes an installation script (`install.sh`) with which we can choose from several installation options available. These options are mainly related to security and deployment. To be more specific, they focus on how the Zope process is going to be launched and how to configure a cluster of Zope Server. As they have a lot to do with the deployment process, we will discuss them in depth in a separate chapter.

So, we will run the installation script with the option `standalone`:

```
$ cd Plone-3.3.5-UnifiedInstaller
$ ./install.sh standalone
```

This action will automatically perform the following tasks:

- Compile Python, and install it in a way that it doesn't interfere with the main system Python interpreter
- Compile and install the required libraries if necessary (zlib, libjpeg, PIL, and so on)
- Create a main directory `Plone` under the home of the current user (that is, `/home/victor/Plone`)
- Install `setuptools` and `zc.buildout`
- Compile and install Zope server
- Install a Zope instance in `Plone/zinstance`
- Configure and install some management scripts in `Plone/zinstance/bin`
- Add a Plone site in the Zope ZODB root called `Plone`
- Set up Zope administrator user account and a *random* password that shows in the console messages during the setup

Once the installation is done, we can start the Zope server by executing the following script:

```
$ ~/Plone/zinstance/bin/plonectl start
```

To stop the server, execute the following script:

```
$ ~/Plone/zinstance/bin/plonectl stop
```

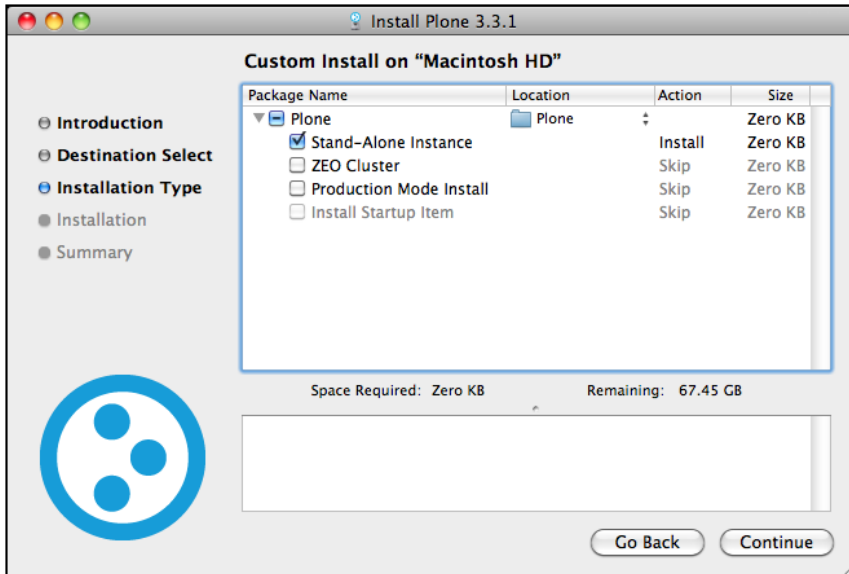
To check its status, execute the following script

```
$ ~/Plone/zinstance/bin/plonectl status
```

For more details and other relevant configuration options, see **Installing Plone 3 with the Unified Installer** at <http://plone.org/documentation/tutorial/installing-plone-3-with-the-unified-installer> in the Plone site **Documentation** section.

Mac OS X

There is also a unified installer for Leopard and Snow Leopard. The installation process is very straightforward and is very similar to the Linux one, but with a graphical user interface assistant. The installer creates a base installation in `Applications/Plone`, although we can choose other target directories. It includes a Mac version of the Plone controller, available at `Applications/Plone/zinstance`.



It's also possible to install Plone in Mac OS X with the Linux unified installer too, but it's recommended to use the one which is platform specific.

Buildout

The module `zc.buildout` (or Buildout—refer to <http://www.buildout.org> and <http://pypi.python.org/pypi/zc.buildout> for more details) has become one of the most extended tools in development environments, production environments, and projects based on Python. As we noted before, from Plone 3.3, all unified installers are based on the software package `zc.buildout`.

Written in Python by Jim Fulton, buildout is an application deployment system that makes heavy use of Python eggs. This tool can build a complete environment, isolated from the rest of the system, with the help of information provided by a configuration file that will comprise the definition of our whole application. In this configuration file (`buildout.cfg`), all components, elements and parts needed by our application, and its related settings, are defined. These parts range from installation of additional software needed by the project and its configuration, to the installation of required modules and their configurations along with the overall configuration and customization of our application.

Thus, we only need `zc.buildout` module (which is a Python egg itself) and the buildout configuration file. Once launched, the `buildout` script *mounts* the entire environment, downloading everything needed, installing, and configuring it as defined in the configuration file.

Although we will not develop software, the buildout approach is the easiest way to install Plone in a robust, methodical, and repeatable way. It is also very easy to install new Plone products through the buildout configuration file.

Distribute, setuptools, and eggs

Distribute extends the capabilities of **distutils**, Python's built-in packaging system. It provides, among other things, the ability to download and install a Python module and all its dependencies, and supports egg packaging format. It includes a powerful script utility called `easy_install` that manages all the eggs installed in the system or current self-contained environments (such as buildout or virtualenv). It also takes care of the latest versions installed, maintains a local cache, and is able to download eggs from online resources or repositories such as the Python Cheese Shop, also known as the **Python Package Index**, or **PyPI** (<http://pypi.python.org>).

Distribute is a fork of the setuptools project. It is intended to replace setuptools as the standard method for working with Python module distributions. Distribute has full backwards compatibility with Setuptools, the code is actively maintained and it offers Python 3 support.

A Python egg is a specific binary distribution format based on ZIP files. An egg also provides project-wide metadata, package specific data, additional Python code, and keeps track of all dependencies the project or module may need.

As someone said, "Eggs are to Python as Jars are to Java". Nowadays, a lot of Python software projects have migrated or they are currently in the process of being eggified. For example, Plone ended its migration to a fully eggified version since Plone 3.2. Zope migrated to a completely eggified version in Zope 2.12. Thanks to distribute, we can find resolve dependencies (and install them too, if needed), build and install eggs by specifying the package name and version, and download them directly from online repositories.

Buildout uses Distribute to setup the full environment needed to run Plone and Zope, defining it in a few lines of code.

PasteScript and ZopeSkel

PasteScript is a library for creating Python project templates that ZopeSkel heavily relies on. All skeletons are available as PasteScript templates and can be used via the `paster` command-line tool. ZopeSkel is a collection of skeletons for quick starting Zope projects. It adds to the list of available PasteScript templates a few skeletons that are useful for starting Zope projects quickly. One of them is the `plone3_buildout` template, and is the best way to create our Plone buildout skeleton. This is the command line:

```
$ paster create -t plone3_buildout [name_of_my_buildout]
```

Then, the `paster` command will ask us some questions:

```
Enter plone_version (Which Plone version to install) ['3.3.4']:
```

We must tell it which Plone version we want to use. To have a look at the latest Plone version released visit <http://plone.org/products/Plone>.

```
Enter zope2_install (Path to Zope 2 installation; leave blank to fetch one) ['']:
```

In case we have to manage a lot of Plone buildouts, we can specify a path to a single Zope 2 installation, in order to have only one Zope shared through several buildout installations and save hard disk space. However, nowadays space is not a problem, and is advisable to have one Zope installation for each buildout installed. Leave this option blank to let buildout fetch one for you.

```
Enter plone_products_install (Path to directory containing Plone products; leave blank to fetch one) ['']:
```

This is relevant only for older Plone versions and in case we are using old-style Plone add-on products. The purpose of this option is to define a centralized old-style add-on products folder for our environment. Leave this option blank.

```
Enter zope_user (Zope root admin user) ['admin']:  
Enter zope_password (Zope root admin password) ['']:
```

The previous code shows the default Zope root administrator user account and its password. The convention in development or testing environments is to use the admin/admin pair. However, we can use whatever we like and its associated password.

The following code defines the Zope server port:

```
Enter http_port (HTTP port) [8080]:
```

There are two Zope server running modes, the first one is related to enabling a debug mode where all possible information and error messages are logged, and the second one is related to security where all unauthorized access to any object is logged.

```
Enter debug_mode (Should debug mode be "on" or "off"?) ['off']:  
Enter verbose_security (Should verbose security be "on" or "off"?)  
['off']:
```

After these brief questions, `paster` will configure the buildout configuration file `buildout.cfg` with the corresponding parameters. We can modify these parameters later to reconfigure the behavior of the buildout configuration.

Running buildout

At this moment we can build our buildout. The first step is telling buildout what version of Python we are going to use in our enclosed environment. This is a very important step as it will determine the default interpreter and all the related modules. We will define it in running the `bootstrap.py` script with the desired Python version:

```
$ python2.4 bootstrap.py
```

We are assuming that a more recent Python version is installed in our system, and that it takes the `python` (with no modifiers) command name, so we have to call the right version with the command name `python2.4`. This will create some new directories and create the `buildout` script. The next step is to run it:

```
$ ./bin/buildout -v
```

Now we are adding the `-v` (verbose) modifier in order to make buildout show all outputs possible. In this log, we can check how buildout builds our environment. It uses the default `buildout.cfg` as a configuration file, although it can be changed through the modifier `-c` (for example, `./bin/buildout -c deployment.cfg`). It's useful to manage different configuration files depending on the task we are performing, for example, we could have a development buildout and use them when needed.

Following are the other useful buildout script command modifiers:

Modifier	Description
<code>-o</code>	Run in off-line mode.
<code>-n</code>	Run in the newest mode. With this setting, which is the default, buildout will try to find the latest versions of distributions available that satisfy its requirements.
<code>-N</code>	Run in non-newest mode. With this setting, buildout will not seek new distributions if installed distributions satisfy its requirements.

Buildout directory structure

Following are the most relevant directories and their contents that we may find inside the buildout directory:

Folder/File	Description
<code>bin</code>	The location of the process control and management scripts.
<code>bin/buildout</code>	The buildout script, used to build the buildout environment.
<code>bin/instance</code>	In a default buildout template, the script used to control the Zope service instance process.
<code>bin/repozo</code>	A script that performs backups and restores our ZODB database.
<code>bin/zopepy</code>	A Python interpreter that has all the eggs and packages that Zope would have during startup. This can be useful for testing purposes.
<code>buildout.cfg</code>	The buildout configuration file.
<code>downloads</code>	If buildout needs to download a file from a network location, it is stored here.
<code>eggs</code>	Buildout downloads all the required eggs inside this directory.
<code>parts</code>	This directory is completely managed by buildout. It contains the installations of Zope server and its instances along with all the additional programs, servers, and libraries managed by buildout. It is overwritten each time buildout is built.
<code>products</code>	Location of the Zope2-style products. If we have to install some of these products, all we have to do is place them here.

Folder/File	Description
src	Directory location of the development eggs we may have in our environment.
var	Default location of the Zope Database, logs, and other data directories.
var/ filestorage	Location of the ZODB database and its index and other related files.
var/ filestorage/ Data.fs	The most important file here, the container of the ZODB.
var/log/ instance.log	The default name of the Zope application message log.
var/log/ instance-Z2. log	The default name of the Zope http (ZServer) access log.

Setting up buildout.cfg

This file is the director of all the buildout processes. It defines what is going to be done, how is it going to be done and, most importantly, which configuration must be set up in all its components.

To complete the description of the buildout process we must introduce one more element, the **buildout recipes**. Recipes are the plugin mechanism provided by buildout to add new functionalities to our environment building. A buildout part (the elements inside the parts directory) is created by each recipe we use. Recipes are always installed as Python eggs. They can be downloaded from a package server, such as the Python Package Index (PyPI), or manually.

We generally define in `buildout.cfg` as many parts as we need and each part is controlled by recipes. For example, there is a recipe to download and install Zope, and another to set up a Zope instance.

We can now study a simple `buildout.cfg` configuration. We will cover it step by step. If we wish, we can open the `buildout.cfg` file supplied with the code bundle for this chapter.

The buildout section

This file has a ini file structure as it divides its sections by the `[]` delimiters. We will explain it bit by bit:

```
[buildout]
parts =
    zope2
    productdistros
    instance
    zopepy
# Change the number here to change the version of Plone being used
extends = http://dist.plone.org/release/3.3.5/versions.cfg
versions = versions
```

First, we define the main `[buildout]` section and then start defining the parts we want to configure. Buildout will search these parts as sections in the configuration file, executing the directives of its corresponding recipes. If it fails to find one of them, buildout will raise an error. Sections not specified in the parts definition will not be executed.

The extends directive

The `extends` directive specifies another file or URL that contains extra configuration directives. In this case, we extend it with a list of the correct egg versions required to run Plone. In the next directive, we tell buildout to use this new section as the versions source. This is an excerpt of the `versions.cfg` file:

```
[versions]
# Buildout infrastructure
plone.recipe.zope2install = 3.2
plone.recipe.zope2instance = 3.6
plone.recipe.zope2zeoserver = 1.4
setuptools = 0.6c11
zc.buildout = 1.4.3
zc.recipe.egg = 1.2.2
# Zope
zope2-url = http://www.zope.org/Products/Zope/2.10.11/Zope-2.10.11-
final.tgz
[...]
```

In this case, it downloads the file from the URL defined in the code and uses the `[versions]` section to extend `buildout.cfg`. We can also see that it specifies the download URL for Zope, in this case, the 2.10.11 version. It will be used ahead in the configuration file.

The find-links directive

The next directive of the [buildout] section is find-links.

```
# Add additional egg download sources here. dist.plone.org contains
archives of Plone packages.
find-links =
    http://dist.plone.org/release/3.3.5
    http://download.zope.org/ppix/
    http://download.zope.org/distribution/
    http://effbot.org/downloads
# Add additional eggs here
eggs =
# Reference any eggs you are developing here, one per line
# e.g.: develop = src/my.package
develop =
```

Here we define the URLs of the online repositories from where we can find and download all the required Python eggs. The Python Package Index (PyPI) URL is implicit and we don't need to define it here. In the next directive, we specify the eggs needed by our application environment. They will be downloaded from the Python Package Index (PyPI) or from the additional `find-links` URLs. The `develop` directive is used during the module development process to declare all the development Python eggs we use in our environment.

The zope2 section

Once the [buildout] section is over, we can find the `zope2` section.

```
[zope2]
recipe = plone.recipe.zope2install
url = ${versions:zope2-url}
```

Here we can see in action the first buildout recipe, `plone.recipe.zope2install`. It will download and install Zope server, and all its libraries and sources in the `zope2` part inside the `parts` directory. Please note the URL directive that points to the [versions] section, `zope2-url` directive. This is a common notation in the configuration file and it's used to refer to other sections directives and configurations.

```
[productdistros]
recipe = plone.recipe.distros
urls =
nested-packages =
version-suffix-packages =
```

Although this is rarely used now, the `plone.recipe.distros` recipe is used to install *Zope2-style products* in our Plone instance. We can declare several download URLs of the products we may want to install in the `urls` directive, and the recipe will download, extract, and place them in the `productdistros` part folder inside the `parts` directory. We can also use nested packages (compound packages including several packages such as the old PloneLDAP). This method of installing packages is deprecated as we can find almost all Plone or Zope products in an eggified form. We can install them as eggs from the `eggs` directive of the `buildout` section. Needless to say, the `parts`' `productdistros` folder will become a part of Zope's environment software path. Otherwise Zope cannot find and recognize them as installable products.

The instance section

Finally the instance section.

```
[instance]
recipe = plone.recipe.zope2instance
zope2-location = ${zope2:location}
user = admin:admin
http-address = 8080
#debug-mode = on
#verbose-security = on
```

This section will benefit from the `plone.recipe.zope2instance` that will install and configure a Zope server instance. As we can see, it contains several vital directives, such as the default Zope administrator user password (only used the first time we launch Zope, at which time it is stored in the ZODB), the http port address, and the two commented lines corresponding to the two special modes available when we run a Zope instance.

```
eggs =
    Plone
    ${buildout:eggs}

zcml =
products =
    ${buildout:directory}/products
    ${productdistros:location}
```

Following is another `eggs` directive. As we can see, it includes the previous `[buildout]` `eggs` directive. We can add additional eggs here, but it is a general practice to specify them all at the top.

The `zcml` directive is used to make available to Zope, eggs that otherwise Zope wouldn't be able to see by itself. Zope is aware of the eggs with the namespace Products, such as `Products.PloneFormGen`, but if this is not the case, we must add a `zcml` directive for that additional egg. This forces Zope to read the `configure.zcml` inside the egg or eggs specified, making it available for its use. We usually call them "slugs".

The `products` directive just tells the recipe where Zope must search for Zope2-style products.

The zoepgy section

The last part is used to configure a Python interpreter called `zoepgy`.

```
[zoepgy]
recipe = zc.recipe.egg
eggs = ${instance:eggs}
interpreter = zoepgy
extra-paths = ${zope2:location}/lib/python
scripts = zoepgy
```

It is a command line interpreter with all the Python path and Zope modules available. It's useful for developing purposes.

There are plenty of more configuration options and directives available for `zc.buildout`. We can find a complete reference in the PyPi's `zc.buildout` web page at <http://pypi.python.org/pypi/zc.buildout>.

We can also find more recipes at PyPi. The following URL may help us find them, although note that not all are Zope or Plone related:

<http://pypi.python.org/pypi?action=browse&show=all&c=512>.

Launching Zope

Finally, the only thing that remains is to launch our freshly installed Zope instance. We make it so by running the `bin/instance` script:

```
$ ./bin/instance fg
```

The `fg` modifier will launch Zope in debug mode, and will show its console in *foreground* mode. The `bin/instance` script has several modifiers, but for now we will only focus on three of them — `start`, `stop`, and `fg`. In fact, it has a complete command line interpreter with a complete help system; we can try it by executing the script alone.

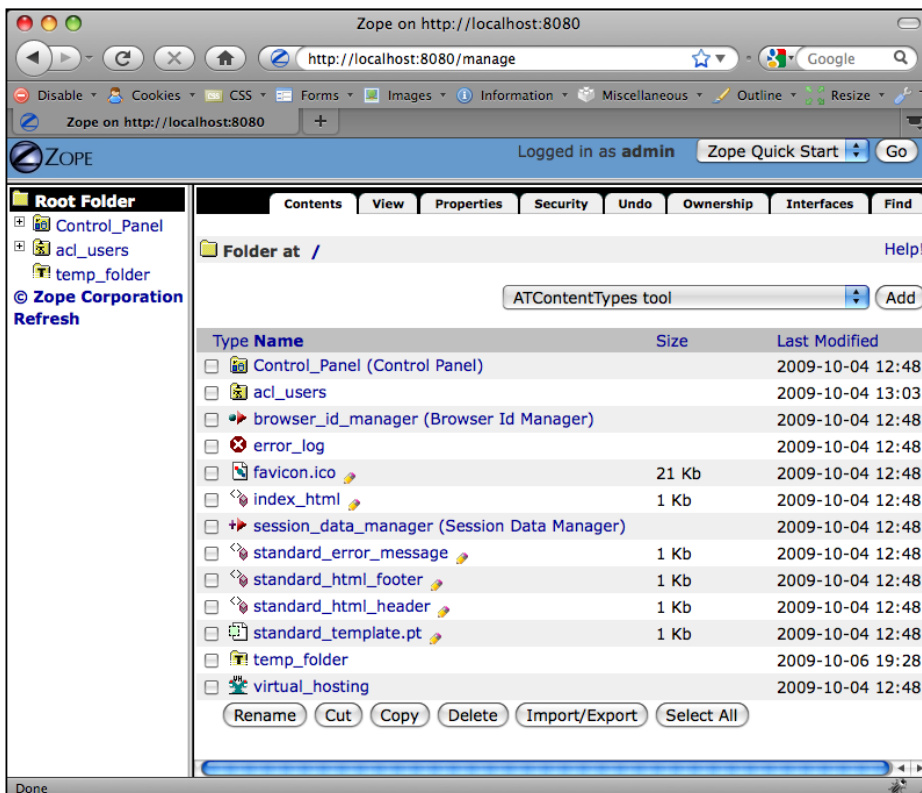
```

$ ./bin/instance fg
/Users/victor/dev/myplonebuildout/parts/instance/bin/runzope -X debug-
mode=on
2009-10-06 19:28:06 INFO ZServer HTTP server started at Tue Oct 6
19:28:06 2009
    Hostname: 0.0.0.0
    Port: 8080
/Users/victor/dev/myplonebuildout/parts/zope2/lib/python/zope/
configuration/xmlconfig.py:323: DeprecationWarning: zope.app.
annotation has moved to zope.annotation. Import of zope.app.annotation
will become unsupported in Zope 3.5
    __import__(arguments[0])
2009-10-06 19:28:14 INFO Zope Ready to handle requests

```

To access the Zope server simply open our favorite browser and type the URL `http://localhost:8080`.

To access the Zope Management Interface (ZMI) we must go to `http://localhost:8080/manage` and log in with the Zope administrator user credentials provided in the buildout creation process. Following is the main ZMI page:



The ZMI is divided into three frames, the top frame containing the headers and a login/logout control, the left column is purely for navigation purposes, and the main frame shows the current element we are inspecting. The default page shown when we access ZMI is the **Root Folder**. All the elements inside the root folder are hierarchically ordered, and as a result Zope Root Folder is the parent of all elements in the ZMI. As in a file system, some of them are **folderish** (this means they can contain other elements) and we can navigate through them. We will learn to use ZMI in depth later in the next chapters. Now is a good moment to play a bit with the environment and make us comfortable with it.

The next thing to do is create a Plone site in the Zope root folder. We manage to do so by selecting **Plone Site** in the drop-down menu at the top right-hand side of the main frame. A brief form is shown where basically we have to set up the name of the new Plone site, and then click on **Add Plone Site**.

Add Plone Site

Enter an ID and click the button below to create a new Plone site.

Id
 (No special characters or spaces)

Title

Description

Extension Profiles
You normally don't need to select anything here unless you have specific reasons and know what you are doing. Leave it blank if you want a default Plone site.

- Workflow Policy Support (CMFPlacefulWorkflow)
- Working Copy Support (Iterate)
- OpenID Authentication Support
- b-org local role plug-in
- NuPlone

NOTE: You may only use ASCII characters for **Id**, **Title**, and **Description** in this form! You can change the values later from the Plone UI, but during creation of a Plone site characters outside the A-Z and numbers range are not allowed.

We can access our Plone site using the URL <http://localhost:8080/mysite>.

Summary

In this chapter, we have learned how to download, install, and run our own instance of Plone, independent of the platform we may have. We have covered the following topics:

- Prerequisites of Plone
- Python and setuptools
- Unified installers
- Buildouts and what they are used for
- Installing Plone using buildout in depth
- Running your first instance of Plone
- First contact with ZMI

In the next chapter, we will take our first steps with Plone from the user's point of view.

3

Managing our Content

Before we advance to building our Plone intranet site, we must first know about Plone, *the application*. We have divided this topic into two chapters: basics and advanced. We will cover Plone basics in this chapter. If one is already familiar with the common use of Plone and its user interface, one can advance to the next chapter. In this chapter, we will learn the key concepts of content management with Plone, and our objective will be to get confident with its interface. We will cover the following topics:

- Plone visual layout structure
- Basic content management
- Content structure
- Content adding and managing
- Default content views
- Managing portlets



In our path to build a complete Plone based intranet, we will begin by referring to our Plone site as "intranet". For this purpose, we will create a site named **intranet**; its URL will be `http://localhost:8080/intranet`, and all screenshots and examples will be based on this assumption.

We will continue from where we left off in the last chapter. We will assume that we have our Plone instance up and running, and we can access it via our favorite web browser.

Plone visual layout structure

It will be a good start if we begin by describing the default Plone layout structure, explaining the locations of every element we will encounter in each page and start calling them by their names.



Since the first versions of Plone, it has shipped with the same old venerable default skin. Although it may seem outdated and old-style design, it's as functional and usable as the first day. In the past few years, there have been many discussions about which default theme should ship with Plone. In the end, it was decided it was time for renovation. Plone 4 will ship with a brand new theme called Sunburst. However, we will use the Plone 3 default theme in the following examples.

A standard Plone site layout is divided into five well-defined regions as follows:

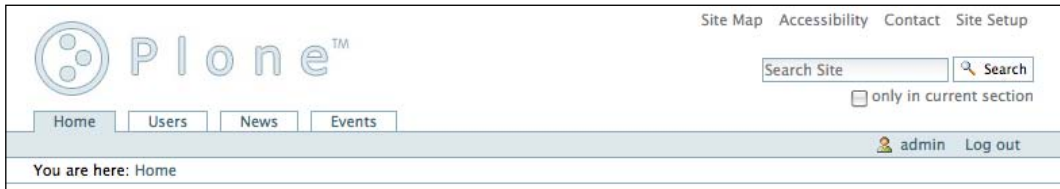
- Header
- Left column
- Content
- Right column
- Footer

The screenshot shows a web browser window displaying the Plone installation page. The page is divided into five regions, each highlighted with a red box and a red label:

- Header:** Contains the Plone logo, navigation links (Home, Users, News, Events), a search bar, and a 'Log in' link.
- Left Column:** Contains a 'Log in' form with fields for 'Login Name' and 'Password', a 'Log in' button, and a link for 'Forgot your password?'.
- Content:** Contains the main body of the page, including the 'Welcome to Plone' message, a 'Get started' section with a list of instructions, and a 'Get comfortable' section with a list of suggestions.
- Right Column:** Contains a calendar for October 2009 and a 'Site Map' link.
- Footer:** Contains the copyright information for Plone CMS and the GNU GPL license, along with links for 'Powered by Plone', 'Valid XHTML', 'Valid CSS', 'Section 508', and 'WCAG'.

Header

The header is the upper most section, and following the common guidelines of a well-designed website, it contains the following elements:



- **Logo:** The logo is a customizable image placed to the left of the header. We will learn how to customize it in *Chapter 12, Theming our Intranet*.
- **Site actions:** The site actions are a collection of links leading to special pages in the site. There are some actions only visible to users with special roles, such as the **Site Setup** action, which can only be accessed by users with the global role *Manager*. The site actions link list is customizable too.
- **Portal tabs:** Portal navigation is automatically built using the top or root level folders and non-folder content. Usually they represent the main sections of our site. Portal tabs are an important navigation tool by themselves and for this reason they are shown in this relevant layout area. By default, all published content located in the root folder is shown as a portal tab.
- **Search box:** The search box is probably one of the most powerful and useful tools available in Plone. It provides a fast and reliable site-wide content search, making navigation in our site more efficient. It is sensitive to the rights owned by the current user. This means that an anonymous user will only receive results about published content, and a logged-in user can only search over the set of content he is allowed to access. The search box also provides a search-ahead with our typing tool called **LiveSearch**.
- **Personal bar:** The personal bar shows information about the current logged-in user, the access to some user tools like the dashboard, the location of the user's personal folder, or the logout action.
- **Breadcrumbs:** The breadcrumbs area shows the path to the current object. This path is formed by the links from the root folder to the current object of the parent folder of an object.

Columns

Located at each side of the content region, they are exclusively reserved for the site portlets. A **portlet** is a portion of information generated for a particular purpose from the existing content in the site, such as to show the last published news or show all the published site events in a calendar view. Sometimes, there are portlets that are not bound to any site content, but they have a special functionality, such as the login portlet or the search portlet.

Content

Content is the most important region of the layout because this is the location where the content of the object we are accessing is shown. The arrangement of the content itself is determined depending on the kind of content type we are accessing, but it usually follows some conventions, at least in Plone's default content types.

Following are the elements you will find in this section:

- Title of the content
- Description (if any)
- Main object's information
- Document actions
- Related content

If we are logged in, we might see some other elements, depending on the roles owned by the current user. From the editing and content management point of view, the most important element in content region is the Plone's **content management tabs**.

Footer

The footer area is located at the bottom of the page, and is the usual place to show general information about our site or organization.

The Plone[®] CMS — Open Source Content Management System is © 2000–2010 by the Plone Foundation et al.
Plone[®] and the Plone logo are registered trademarks of the Plone Foundation. Distributed under the GNU GPL license.

[Powered by Plone](#) [Valid XHTML](#) [Valid CSS](#) [Section 508](#) [WCAG](#)

By default, it shows general information about Plone, its license type, the standards it meets, and the Plone Foundation.

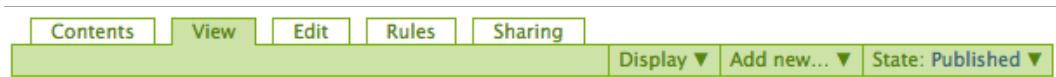
Anonymous versus logged in

The previous screenshot showed the default Plone front page, as an anonymous user would see it. If logged in as a Zope administrator, we may have seen some differences. Depending on the rights assigned to our user, we will notice some user interface differences, such as some additional links (for example, site setup link in the site actions) or Plone's content management tabs.

Content management tabs

One of Plone's more powerful features is the in-site editing; in other words, the user interface for viewing and editing content are the same. This is a very important feature in an intranet site, because we always have the same interface even if we are just reading content as consumers, or we are creating or editing new content as authors. In an intranet, we constantly perform both these things simultaneously.

The Plone content management tabs will help us perform all the possible tasks over the site content. If we are logged in as administrator, we will have full power to do everything and we will also see the tabs located over the content region as follows:



This element contains several tabs and drop-down menus that allow us to perform several actions over our content, and switch between several useful views. Sometimes, people from the Plone world refer to them as **content actions** and **content views**.

By default, we can see five content tabs corresponding to five content views, as follows:

- **View:** The default view of the selected content.
- **Contents:** If the selected content is folderish (it can contain other content objects), then the **Contents** tab shows a view with all the elements that it contains. The contained contents are showed in a list format that can be useful to make bulk actions (such as to rename, copy, change status, among others) on them.
- **Edit:** The content edit form.
- **Rules:** The content rules configuration form. Content rules are predefined actions that apply to the content inside a folder when a specific event is triggered. For example, we can send an e-mail each time a new content is added to a folder using content rules. More on this can be learnt in *Chapter 11, Content Rules, Syndication, and Advanced Features*.

- **Sharing:** This is a form that allows us to manage the selected content permissions. We have the ability to define fine-grained permissions for a user, or group of users, to allow them to interact with our content from this form.

Following are the content actions by default:

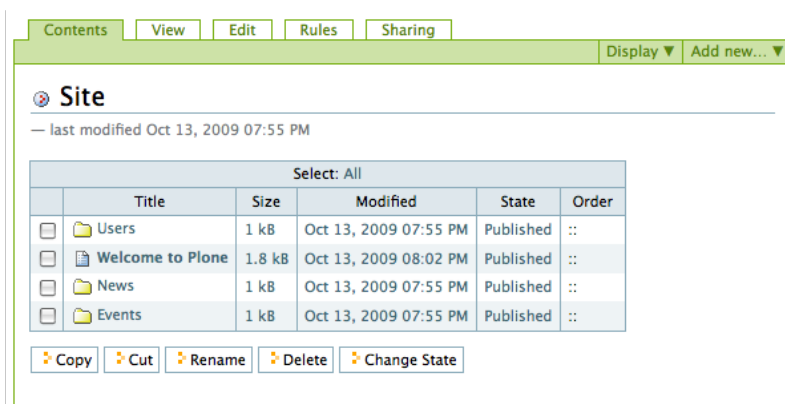
- **Display:** Allows us to choose the selected content default view
- **Add new...:** A drop-down menu for creating new content inside this folder
- **State:** Let us change the current content state

Due to Zope's fine-grained permissions, we will be able to define which users have the rights to perform each action or have access to these views.

Content structure

As we already pointed out, Zope's object structure is hierarchical, which means that a typical Zope site is composed of objects that contain other objects (such as, *ad infinitum*). URLs map naturally to objects in the hierarchical Zope environment based on their names. Mapping URLs to objects isn't a new idea; web servers such as Apache and Microsoft's IIS do the same thing. They translate URLs to files and directories in a filesystem. Zope similarly maps URLs onto objects inside the ZODB. A Zope object's URL is based on its path. It is composed of the IDs of its containing folders and the object's ID, separated by slash characters.

The Plone site's root, which is the Plone site object itself, is mapped to the site's root URL (for example, `http://servername/mysite`). We can see the default page assigned for the Plone site's root. This is because we can assign an object as a default view of any folderish object, such as folders or the Plone site. In the Plone site's root, if we click on the **Contents** tab, we will see all the objects it contains, as shown in the following screenshot:



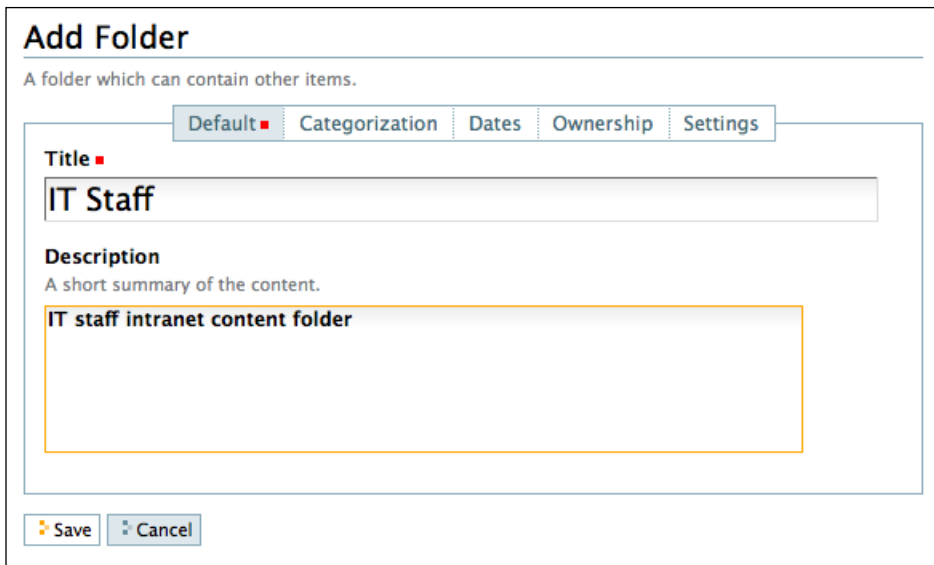
At the time of creation, a Plone site contains a few objects by default, such as:

- **Users:** A special folder that can contain a user's personal content and a form for searching site members. Since Plone 3.3, the user's personal folder is not activated by default; we have to manually enable this feature from Plone's control panel.
- **Welcome to Plone:** The default site main page (note the bold font used for displaying it).
- **News and Events:** These are the folders containing news and event content types and the views to display them.

From here, we can navigate thorough the site objects as if it was a filesystem. Note how every object is mapped uniquely with its own URL, and how the path is formed depending on the object's hierarchy.

Adding content

Now, it's time to add some new content to our intranet. Folders are normally used to organize information in our Plone site. First, we must go to the folder where we want to create the new content, for example the root intranet folder. Then from the drop-down **Add new...** menu select **Folder**.



Add Folder

A folder which can contain other items.

Default | Categorization | Dates | Ownership | Settings

Title

IT Staff

Description

A short summary of the content.

IT staff intranet content folder

Save Cancel

In the example, as shown in the previous screenshot, we are creating the folder that will contain the IT staff intranet information. In this simple form, we will assign a **Title** and a **Description** for this folder. Now click on **Save** and let's see the results. We have created the desired folder in the intranet's root folder, and we can notice the following three important things:

- A portal tab is created automatically each time we create some content in the root folder, and we have rights to access it.
- Our content default state is private, so anonymous users can't access this object until we change its state and publish it.
- The new content's URL is composed of the intranet root URL and the **ID** of the new object which is the *normalized* version of the **Title** assigned to the object on creation (<http://localhost:8080/intranet/it-staff>).

Next, let's add some content to our brand new folder. Be sure that we are in the folder **IT Staff** and select **Page** from the **Add new...** drop-down menu.

Add Page

A page in the site. Can contain rich text.

Default ■ Categorization Dates Ownership Settings

Title ■

My first page

Description

A short summary of the content.

Here goes the description

Body Text

B *I* [List icons] [Link icon] [Image icon] [HTML icon]

Normal paragraph

My first rich text content.

Change note

Enter a comment that describes the changes you made.

Save Cancel

The form asks us to enter the new page's **Title** and **Description**, and two more fields as follows:

- **Body Text:** A rich text field powered by Kupu, which is a basic rich text editor
- **Change note:** Helps users to keep track of the changes made to the content

The page content type features a rich text editor widget called Kupu. It enables the most basic text editing features, such as text formatting or applying predefined styles. It also has a feature that allows us to edit the resultant HTML directly for more advanced editing.



Plone 4 will switch to a more modern rich text editor called TinyMCE due to its modular structure, small core, large user base, and because it is actively developed and easily themed.

Standard Plone content types

Plone ships out-of-the-box with eight standard content types, all of them feature the Title and Description fields, as they are part of the **Dublin Core** metadata set of fields. Following is a brief explanation of each of them:

Content types	Description
Folder	A folderish item, it can contain other content types. Used to organize and group content.
Page	The standard rich text page content type.
Event	Holds information about an event, such as its location, start and end date, details, attendees, and online presence. If published, they will show up in the calendar and event portlet.
File	Creates an object that holds a file.
Image	Creates an object that holds an image with the engine to show it properly.
Link	An object that points out to a link (internal or external).
News item	Holds information about a news item, such as its details and an additional embedded image that will show up in some site's views. If published, they will show up in the news portlet.
Collection	A powerful content type, it features the ability to show the results of a user-defined query to the database. This query can be of the type: <i>Show me all pages created by user X last month, or show me the news with the keyword Y published during this year.</i>

Content metadata

The **Dublin Core** metadata element set is a standard for cross-domain information resource description. It defines conventions for describing things online in ways that make them easy to find. Dublin Core is widely used to describe digital materials such as video, sound, image, text, and composite media such as web pages.

Plone's standard content types are Dublin Core compliant, as they have all the needed metadata attributes defined in the standard. We can access them while editing content via the **metadata tabs**.

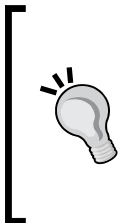


The following tabs allow us to set and modify the content's metadata:

- **Categorization:** Here we can define keywords (tags or labels), location, and content's language
- **Dates:** The content's publishing date and the content's expiration date
- **Ownership:** Author related metadata

Content settings

Content stores information about its functional properties too. These properties are located in the **Settings** tab. It's also used to set some important behavior about content items. In the case of folder contents, we can allow comments, or enable a special *next/previous* navigation on content items contained in that folder. A very useful setting would be the ability to exclude the item from the standard site navigation (portal tabs and the navigation portlet).



The Exclude from navigation setting

This is a useful setting under the **Settings** tab. When we have a folder to contain support images for a page, probably, we wouldn't have to show it in the navigation portlet. Use this setting to avoid displaying it on the site's navigation portlet. For published elements in the root folder, use it if we don't want Plone to show them automatically in the portal tabs.

Managing content

There will be times when we will need to take some management actions on our content, such as moving, copying, or renaming it. We can achieve this by using the controls located at the **Contents** view.

Select: All					
	Title	Size	Modified	State	Order
<input type="checkbox"/>	Users	1 kB	Oct 13, 2009 07:55 PM	Published	::
<input type="checkbox"/>	Welcome to Plone	1.8 kB	Oct 13, 2009 08:02 PM	Published	::
<input type="checkbox"/>	News	1 kB	Oct 13, 2009 07:55 PM	Published	::
<input type="checkbox"/>	Events	1 kB	Oct 13, 2009 07:55 PM	Published	::
<input type="checkbox"/>	IT Staff	1 kB	Oct 18, 2009 01:33 PM	Private	::

These controls provide basic control over content location, such as copy, move, and paste. It also allows us to delete contents and change its state.

We can rename elements as well from here; in this case a rename element form will show up allowing us to change the ID of the element and its **Title**. We must be careful when renaming IDs, because they should contain only ASCII characters and should not have any space, as it will become a part of the object's URL. For example, "it staff" as ID is not acceptable, as it contains a space. A correct ID would be "it-staff". Notice the checkboxes on the left side of each object, which provide multiple object selection.

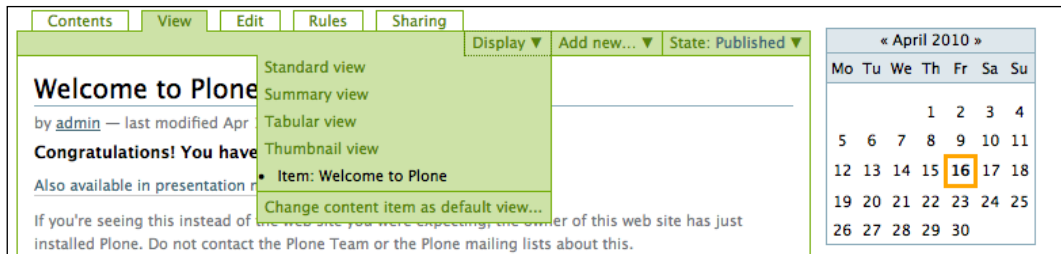
We can also define the position of the object in the folder, by using the column **Order**. We can drag and drop the elements (drag them from the :: symbol) until they are positioned as we wish.

These controls are also available for each object from the **Actions** drop-down menu. In this case, the action is related to the current object.

Displaying views

Folders have the ability to display a default view every time we access it. From the drop-down menu **Display**, we can choose between four standard views or select an existing content item as the default folder view. For example, Plone's default main page is a page that lives in the root folder, configured as the default view of the Plone site object.

Following are the standard views of Plone's folder:



- **Standard view:** The default folder view
- **Summary view:** Displays the contained elements **Title** and **Description** and a **Read More** link, such as in a blog view
- **Tabular view:** Shows the contents of that folder in a table, such as in the Contents view
- **Thumbnail view:** If we have an images folder, this view will show the thumbnails of the images in a gallery-like view

Needless to say, assigning a default view to a folder is always more elegant than leaving it at its default view, if it is a structural folder or a first level folder.

Managing portlets

We can manage portlets by accessing the portlet manage form. The **Manage portlets** link is located at the bottom of an existing portlet or at the bottom of the content region.

Manage portlets for "IT Staff"

Return

The portlet columns will now display only those portlets explicitly assigned in this context. Use the buttons on each portlet to move them up or down, delete or edit them. To add a new portlet, use the drop-down list at the top of the column.

If you wish to block or unblock certain categories of portlets, you can do so using the drop-down boxes.

Parent portlets:
Do not block

Group portlets:
Use parent settings

Content type portlets:
Use parent settings

Save settings

Portlets assigned here

Inherited portlets

Review list

News

Events

Calendar

Block/unblock portlets

Parent portlets:
Do not block

Group portlets:
Use parent settings

Content type portlets:
Use parent settings

Save settings

Plone's portlet engine features the assignment of a portlet to any object of the site. The assignment is inherited, which means if we assign a portlet to the root folder, then all its children objects will have it assigned. However, this inheritance can be cancelled. Following is the list of portlets available by default:

Portlets	Description
Calendar	The default calendar view, shows the published portal events in it
Classic	For compatibility reasons, it is also possible to define an old-style Plone 2.x portlet
Collection	Shows a predefined result from a previously created collection object
Events	Shows the published events in a list
Login	The standard login portlet, it provides the authentication form
Navigation	Shows the position of the current item in folder hierarchy along with the other items in the same level
News	Shows the published news in a list

Portlets	Description
RSS Feed	Allows us to configure an RSS feed and shows the contents of the feed in it
Recent items	Shows the recently modified items in the site
Review list	Shows a list of elements that haven't yet been reviewed (state pending) by the current user
Search	Shows the search form inside a portlet
Static text	Shows user-defined static rich text within a portlet

We can add them in each column region from the drop-down menu **Add portlet...** We can see the already assigned portlets and the inherited portlets as well. We have the option to block the parent portlets, and do the same with the group portlets or the content-type portlets.

The parent portlets are the inherited portlets from the parent folder. Another portlet engine feature is to assign portlets depending on the current user group membership. We can also assign portlets to a content type, so each time this content is created the assigned portlets will show automatically.

We can order them too, via the arrow icons located at the right-hand side of each portlet. We can also unassign them using the red cross icon.

Summary

We have learned Plone's basic content structure, creation, and administration. This chapter has covered the following topics:

- Plone visual layout structure
- Basic content management
- Content structure
- Content adding and managing
- Default content views
- Managing portlets

There are more things we should know about, such as **Sharing** view or **Rules** view, but we will cover them in the next chapters. This section outlines only the basics and is not to convey in-depth information about Plone, but gives sufficient information to have a broad knowledge about Plone's content management.

In the next chapter, we will learn about Plone's configuration and advanced setup, and how to install third party products.

4

Configuring our Site

This is the moment to move on and learn some more advanced subjects on Plone. Maybe you are thinking that, probably, there are lot of more Plone basic stuff we didn't cover, but don't worry; we will cover them sooner or later as we unfold different topics in the book.

We have covered how to add content, how to structure it in our intranet, and perform basic operations on it. We also learnt that all contents can have state, and that state controls a user's rights, depending on his roles and permissions. State also allows us to define the content's life with more or less complexity. We already know how to assign portlets to content and how to define default views to it.

Now it's time to learn how to configure our site and the tools we have available in order to succeed in this task. We will make our first stop at Plone control panel, then we will learn about Zope Management Interface (ZMI), and at the end of the chapter we will know how to install third party Plone products via `zc.buildout`. This chapter could also be considered as a reference manual for Plone control panel and ZMI.

This chapter will cover the following topics:

- Plone control panel
- Zope Management Interface (ZMI)
- Zope control panel
- Database management
- Products management
- Plone site from ZMI point of view
- Installing add-on products

Plone control panel

Plone provides a complete configuration tool through its UI, the **Plone control panel**. It is composed of small configuration elements called **configlets**, each of them focussing on one Plone feature. Users can access it if they have the *Manage portal* permission. We will take special care of those configlets that will help us maximize our site as an intranet.

We can have access to the control panel thorough the **Site Setup** action, or by adding `plone_control_panel` to the root URL address of our site. This is what it shows:

Site Setup

Configuration area for Plone and add-on Products.

Warning You have not configured a mail host or a site 'From' address, various features including contact forms, email notification and password reset will not work. Go to the [Mail control panel](#) to fix this.

Plone Configuration

Add-on Products	Mail	Themes
Calendar	Maintenance	Types
Collection	Markup	Users and Groups
Content Rules	Navigation	Visual editor
Errors	Search	Zope Management Interface
HTML Filtering	Security	
Language	Site	

Add-on Product Configuration

No preference panels available.

Plone Version Overview

- Plone 3.3.1
- CMF 2.1.2
- Zope (Zope 2.10.9-final, python 2.4.6, darwin)
- Python 2.4.6 (#1, Sep 20 2009, 13:02:49) [GCC 4.0.1 (Apple Inc. build 5490)]
- PIL 1.1.6

Note: You are running in "debug mode". This mode is intended for sites that are under development. This allows many configuration changes to be immediately visible, but will make your site run more slowly. To turn off debug mode, edit your zope.conf file to say 'debug-mode off' — then restart the server process.

The Plone control panel is divided into the following three parts:

- **Plone Configuration** on the top
- Add-on Configuration items in the middle
- Version information summary about Plone and its dependencies at the bottom.

The configlets are usually a form or set of forms that help us configure one aspect of Plone or a Plone add-on product.

The first thing we will notice is the warning message that informs us that no mail host is configured for this site. For this reason the features involving mail sending, such as contact forms, email notifications, and password resets will not work until we correct this. Let's start by fixing it.

Mail control panel

We must configure the essential data to provide mail sending capabilities to our intranet. If we host our own mail service or use an external provider, we can use our SMTP configuration to set up Plone mail. This form has **SMTP server**, **SMTP port**, **site 'From' name**, and **site 'From' address** as required fields. In case our SMTP server requires authentication, we can also specify **Extended SMTP (ESMTP)** username and password.

The **site 'From' name** and **site 'From' address** fields are used as the name and address of the sender of the generated mail. The **site 'From' address** is also used as the destination address on the site-wide contact form.

Check the setting by opening the site's contact form (we can access it from the **contact** link located at the site's header) and send a comment to the administrator.

Site

This is an important configlet, as here we can define some special attributes of our intranet. They are important because of their visibility, such as the site's title and description. They also enable special features such as optimizing how search engines crawl our site. Following are the most relevant attributes:

Attribute	Description
Site title	The title of the site that shows up in the title bar of browsers, syndication feeds, and so on.
Site description	It's not normally visible to users but it's available in syndicated content, and search engines index it.
Show 'Short Name' on content?	Allows users to edit the 'short name' content identifiers, which forms the URL part of item's address. It's easier for users to change this attribute from the rename content action.
Enable link integrity checks	Determines if users should get warnings when they delete or move content that is linked from inside the site.

Attribute	Description
Enable External Editor feature	Determines whether the external editor feature is enabled. This feature requires a special client-side application installed. The users also have to enable this in their preferences.
Expose sitemap.xml.gz in the portal root	Exposes our content as a file according to the <code>sitemaps.org</code> standard. We can submit this to compliant search engines such as Google, Yahoo, and Microsoft. It allows these search engines to crawl your site more intelligently.
JavaScript for web statistics support	For enabling web statistics support from external providers (for example, Google Analytics). Paste in this text field the code snippets provided. It will be included in the rendered HTML as entered near the end of every page the site generates.

As we can see, some of them have to be carefully configured. The site title and description are used in almost every aspect of the process of publishing our site to the world.

Nowadays it is common to rely on external providers (for example, Google Analytics) to add advanced web statistics to a site or intranet, and it often involves the use of a JavaScript snippet inserted in every page of the site. As the snippet runs each time a page is viewed, it will track site statistics even if it's an intranet. Plone allows us to include the tracking snippet without modifying the templates. We only have to paste the snippet code in the text box.

The link integrity check is a Plone feature that keeps track of all internal links we insert in our pages. If we move or delete some content, Plone checks whether this action will break any link to that moved or deleted content. If it does, then Plone shows us a warning message and the list of items containing a potential broken link. We can edit these items and remove the references first.

Users and groups

This is another important configlet in which we find three forms for managing users and groups of our intranet, and for managing global settings related to users and groups. We can create new users or groups, assign roles to them, and delete them. These roles are global; it means that they are valid for the entire intranet. In case we have a lot of users, a search form is also available. The **Settings** tab has two configurations destined to optimize the handling of a large installment of users and groups in our intranet.



We will cover in depth users, groups, roles, and all related settings located in this configlet in *Chapter 5, Managing users, Groups, Roles, and Permissions*.

Security

This allows us to configure some security aspects of the intranet. Some of them enable relevant features to an intranet. Let's see them:

- **Enable self-registration:** Allows users to register themselves on the site. If not selected, only site managers can add new users.
- **Let users select their own passwords:** If this is not selected, passwords will be automatically generated and mailed to users, which verifies that they have entered a valid e-mail address. If this and the previous setting are selected, managers cannot assign passwords directly and they get mailed to users too. However, there is a time limit within which the user has to confirm the mailed password.
- **Enable User Folders:** If selected home folders are created for the first time, the user can log in and create content in these folders.

Plone has the capability of allowing new users to register themselves using a registration form and follow a predefined registration process. This is the first moment where we have to decide what kind of intranet we want:

- An intranet where only managers can add and manage users
- An intranet that allows anonymous users to register as intranet members and participate in our intranet if we allow them to do so (forums, forms, and so on).

Another great feature of Plone is the users' folders. These are considered as a personal place where users can create and manage content according to their will. It can be used as a personal content container, the place of the user blog, or even as a temporary container before placing content in its final destination (for example, private content). For security reasons, this feature is disabled by default.

Types

From this configlet we can choose workflow, versioning, and visibility settings of content types used in our intranet. In an out-of-the-box Plone site, all content types have the default workflow applied, except for files and images, which don't have any workflow by default. If not set, the default workflow setting is applied to any content type in the site, although some third party products may define and use their own workflow. **Simple Publication Workflow** is the default workflow. We can define the default workflow of all content types assigned to the default setting by changing it using the drop-down box **New workflow**. We can override the use of the default setting for a particular content type by selecting it from the main drop-down box and changing the workflow for it.

If we select a content type, we can change other type-related behavior, such as:

- **Globally addable:** Enabled by default for all types. If we don't want the type to appear in the **Add new...** drop-down box, then uncheck it.
- **Allow comments:** If enabled, comments will be allowed for the selected content type.
- **Visible in searches:** The type will not be shown in searches.
- **Versioning policy:** To enable or disable the versioning feature.


We should be aware that changing the default workflow for a type could take a while, and may slow down the site significantly while the content is being updated to the new setting, so use it carefully in production.

Add-on products

This configlet lists all the available add-on products for the intranet. Since we didn't install any additional add-on products, we can see the available products in the list that Plone can install in the site. The most significant ones are:

- **OpenID authentication support:** Users can log into the site with its OpenID username and password
- **Workflow policy support:** Allows us to define workflow policies that define content type to workflow mappings that can be applied to any sub folder of our site
- **Working copy support:** Adds the capability of staging content (modifying an item while the original one is in place)


We can install any product by selecting it and clicking on the **Install** button. Then a list of already installed products will be shown below the available ones.

 At the end of this chapter we will learn how to install new products via buildout.

Content rules


Since Plone 3, we can define a set of rules that apply a certain action to content when an event is triggered. By default, only users with the manager role can add and modify these rules. Content rules can be defined over containers and the rule applies to its contents.

It's possible to enable or disable this feature globally and manage all the rules defined in our intranet from this form.

 We will cover content rules in depth in *Chapter 11, Content Rules, Syndication, and Advanced Features*.

Maintenance

The ZODB is a transactional database, this means that it stores every change made in the database as a transaction. This enables some useful features, such as the undo action. In Plone, we can undo an undesired action by reverting the content back to its original state, prior to the action.

 This feature is disabled by default in the Plone UI as it only works under certain conditions. For example, in the case that the change we are requesting to undo has been overwritten by a more recent change, it's not possible to undo that change. However, it's a very useful feature and there is a high chance of success when executing this action. For this reason, having it available on our intranet is always recommended. We can enable it via ZMI as follows: go to **portal_actions**, click on **user** section, and then click on **undo**. Then, enable this action in the **Visible?** box. More information on the Zope Management Interface will be available later in this chapter.

Although it's a nice feature, this also means that the database grows, and grows not only with the real data, but also with all the transactions made. Sometimes we must perform some special maintenance action, such as **packing** over the ZODB. In this configlet, we can find the form that will help us pack a site database. In this form, the current database size is shown along with a field containing the days of history that we want to preserve. This number indicates how many days of undo history we want to keep in the database. The recommended value for a production site is seven days. Packing the database preserving zero days is suitable only for development and non-live sites. Packing leaves a copy of the old database on the server in case the packing process goes wrong. The name of this database is `Data.fs.pack`. So, when we pack our site's database we need free disk space at least the size of our database. We should pack our database regularly to avoid the database growing uncontrollably.

Obviously, the consequence of packing is the removal of all transactions until the day specified, resulting in our inability to undo transactions beyond that date. It is unrelated to versioning, so even if we pack the database, the history of the content changes will be kept.

The configlet also holds the control to restart or stop the Zope server process via the Web.

Errors

This page lists the exceptions and errors that have occurred in the site recently. We can configure how many exceptions should be kept and whether these exceptions should be copied to Zope's event log file(s). We can also refresh, clear displayed entries, or show all entries. We can configure the number of exceptions to keep, and select the exception types as well. If we have problems in our site, it is good practice to find out what is happening and troubleshoot to find possible solutions.

HTML filtering

By default, Plone filters from all rich text fields, HTML tags that are considered to be security risks. It also removes any tags not defined in XHTML specification. However, we can override these settings by removing the filters from this page. The filtered tags are called **nasty tags**, and are listed at the top of the form. There are other tags that are stripped only when saving or rendering, but the content is preserved.



Adding flash to our site's content

It's a common use case to have the need of inserting FLV or shockwave Flash content into our content. In order to allow it, we must remove the tags `param` and `object` from the list of stripped tags, otherwise our flash content will not be shown. Enable it only if you trust your intranet contributor users, as doing this is a bad idea if untrusted users can create content. This limitation is about to change, after the introduction of the new HTML5 tag `<video>`, which will help to standardize the display of video snippets on the Web.

Notice that there are three tabs – one for tags, one for attributes, and one for styles. We can also define custom tags, add a combination of attributes, or add filtered classes if we wish.



Filtering tags, attributes, and styles have another important purpose: to maintain the style and keep the look and feel consistent across the content of our site. Setting the filters properly prevents us from having ugly content such as purple text, weird table sizes, and blinking yellow phosphorescent text inside intranet text contents. Believe me, we don't want this on our intranet; keep our intranet elegant by keeping it neat and clean.

Language

We can define from here the default language used in our intranet. Although it is not common, we might need to support more than one language. In this case, we should install a third-party add-on—**LinguaPlone**. It provides an engine for multilingual content creation and support.



We can speed up Plone's startup time and reduce RAM usage by disabling languages we aren't using by adding one line in the instance section of our `buildout.cfg` file.

```
[instance]
  recipe = plone.recipe.zope2instance
  ...
  environment-vars =
    PTS_LANGUAGES en, es, ca
```



Plone is translated into more than 40 languages, thanks to the **Placeless Translation Service (PTS)**. When starting up, all of these translation files are loaded and information about them is stored in the ZODB. This takes time and memory. By setting the `PTS_LANGUAGES` environment variable with the languages that our site will use, we will load only those languages and hence save startup time and resources.

Markup

In this configlet, we can choose the default input text format for newly created content objects. The default setting is `text/html`, but we can choose between other input formats such as `reStructuredText`; however, the WYSIWYG editor won't work then. We can also choose which formats will be available for users as an alternative to the default setting.

Wiki formatting

We can also enable wiki behavior on the primary text area of the content types that currently support this feature: Page, Event, and News item. Enabling it will provide usable wiki-like linking and content creation for the supported content types. All links are specified by enclosing content text in double parentheses: ((This is a link)).

A link specified as a wiki link will point out to a content object in the rendering process following one of these conditions:

- A matching *ID* within the same folder
- A matching *title* within the same folder
- A matching *ID* somewhere else in the site
- A matching *title* somewhere else in the site

An unresolved link will generate an add link in the form of a plus sign + after the link text. Clicking on this link will create a new piece of content of the same type as the document being viewed, in the same folder.

Navigation

This configlet holds several important settings, all related to navigation and how navigation is constructed in our intranet.

- **Automatically generate tabs:** The first level navigation items (global section items) are automatically added from the content items created at root site level. This setting enables or disables this feature. We can turn this off if we prefer to construct this part manually via ZMI **portal_actions**.
- **Generate tabs for items other than folders:** By default, any root folder item will be displayed as a global section; if we turn this off, only folder items will be shown.
- **Displayed content types:** The content items to be displayed in the navigation tree and site map.
- **Filter on workflow state:** Only the items with these workflow states will be shown.

Search

For refining the search settings on our intranet, the following configurations are available:

- **Enable LiveSearch:** Enables the LiveSearch feature, which shows live results as we type, if the browser supports JavaScript.
- **Define the types to be shown in the site and searched:** Defines the content types that should be searched and be available in the user interface of the site.

Theme

Related to site theme and skin we can find some interesting controls, such as the following:

- **Default theme:** A control that enables us to change the theme of our site, provided that we have previously installed at least one theme package. For example, NuPlone theme, shipped out-of-the-box with Plone 3. Plone 4 ships with Sunburst and Classic themes.
- **Mark external links:** If enabled, all external links will be marked with link type-specific icons. If disabled, the next setting will have no effect.
- **External links open in new window:** If enabled, all external links in the text content region will open in a new window.
- **Show content type icons:** If disabled, the content icons in folder listings and portlets won't be visible.

Zope Management Interface

We've just learnt the most relevant configlets of the Plone control panel, but to continue completing our Plone understanding we will have to leave the comfort of Plone UI and discover what is behind the scene. To see what is happening under the hood we need to switch to the Zope point of view through its management tool, the **Zope Management Interface** or **ZMI**.

We have already had a first experience with ZMI, as it's necessary to access it if we want to create a new Plone site. This is going to change, since Plone 4 provides a special UI to create our first Plone site out-of-the-box.

To access the ZMI we only have to go to the following URL:
`http://localhost:8080/manage`

Type	Name	Size	Last Modified
<input type="checkbox"/>	Control_Panel (Control Panel)		2009-10-04 12:48
<input type="checkbox"/>	acl_users		2009-10-04 13:03
<input type="checkbox"/>	browser_id_manager (Browser Id Manager)		2009-10-04 12:48
<input type="checkbox"/>	error_log		2009-10-04 12:48
<input type="checkbox"/>	favicon.ico	21 Kb	2009-10-04 12:48
<input type="checkbox"/>	index_html	1 Kb	2009-10-04 12:48
<input type="checkbox"/>	intranet (My Intranet)		2009-11-07 19:22
<input type="checkbox"/>	session_data_manager (Session Data Manager)		2009-10-04 12:48
<input type="checkbox"/>	standard_error_message	1 Kb	2009-10-04 12:48
<input type="checkbox"/>	standard_html_footer	1 Kb	2009-10-04 12:48
<input type="checkbox"/>	standard_html_header	1 Kb	2009-10-04 12:48
<input type="checkbox"/>	standard_template.pt	1 Kb	2009-10-04 12:48
<input type="checkbox"/>	temp_folder		2009-11-14 10:47
<input type="checkbox"/>	virtual_hosting		2009-10-04 12:48

This is the root of Zope where we can find all the content objects stored in the ZODB, and all the tools and other objects needed by Zope. We will cover the most relevant ones in the next few lines.

Notice that we can access ZMI from any content object in our intranet if we append `/manage` to the full URL of that particular object. It will show the ZMI point of view of that object, for example, it will suffice to point to this URL `http://localhost:8080/intranet/manage` in order to access the ZMI point of view of our intranet directly.

Control panel

The ZMI provides access to system and Zope's information and management functions, such as database and products management. We will also find version information and relevant information about Zope's features and installation. There is a control to restart or shutdown the Zope server.

Database management


We should find information about Zope's database(s) that are currently attached to it. By default, there's only one ZODB attached to our Zope server, but we can attach several databases to it. Although we can see two databases attached, **main** and **temporary**, the latter is a database stored in memory used for storing objects temporarily (for example, session information). If we follow the **main** database link, we will get access to the database management view which holds several tabs where we can find database information, such as its registered activity; we can manage its cache or flush it, and there are other actions that will empty all the database cache.

Several ZODB attached to one Zope (or ZEO) server



The scenario in which we need to connect several databases to the same Zope (or ZEO) server is not uncommon at all. What will be its purpose? Easy, as Julius Caesar said: *Divide et vinces*, divide and conquer. It's easier to maintain several small databases than one large database. If we have several big sites or intranets and we still want to use the same Zope (or ZEO) server, then it's better to assign each of them to one attached database rather than having one large database. Another case would be having a single big site and span several sections across several databases. Having medium sized databases is always better than a single big one; it's easier to manipulate them, to make backups, and restore them. More information about deployment is available in *Chapter 13, Deploying our Intranet*.

From the default view of this section, we can access the maintenance parameters of the database:

 Database Management at [/Control_Panel/Database/main](#) Help!

The Database Manager allows you to view database status information. It also allows you to perform maintenance tasks such as database packing and cache management.

Database Location /Users/victor/dev/myplonebuildout/var/filestorage/Data.fs

Database Size 12.3M

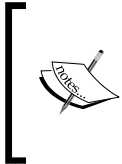
Click **pack** to pack the Zope database, removing previous revisions of objects that are older than days. **Pack**



We have already noticed that Plone provides a control panel UI to access these controls. Obviously, Plone exposes through its control panel UI settings, tools and attributes that already exist in the ZMI. Plone eases the access to all these objects that otherwise would be available only via the ZMI.

Product management

Every time we start Zope server, it does an inventory of all the Zope Products and modules that Zope sees in its software path. Then, this inventory is stored in the ZODB and we can access it in this section. From here, we can't install products in Zope or in Plone. All we can see is the list of products, its versions, and its general information available at the start. We can install new products from the Plone control panel configlet **Add-on Products** or from the **portal_quickinstaller** ZMI tool.



If we delete a product or module of the Zope's software path it's possible to see a warning message in the error log in the server start process. This issue can be solved by deleting the ZODB entry in this section via the controls at the bottom of this section. This is no longer necessary for Zope 2.12 and Plone 4.

Placeless translation service

Zope provides a very good translation service. This service works with `gettext` standard files (`.po` files) that are processed and compiled (in `gettext .mo` files stored in the filesystem) in the server start process. The result of this compilation is saved in a catalog in the ZODB and appears in this section. Here we can manage each `i18n` record, and the most important, we can force its reload in case we change some translation in them.



We can reload them by following the link of the desired record and then pressing the **Reload** button. If we want to reload all the `i18n` records, then we must delete all of them using the bottom controls and then restart Zope.

Plone site—ZMI point of view

When accessing the Plone site from the ZMI point of view we can see a lot of objects, such as content objects, tools, configuration objects, and basic services used by the Plone site. All of them are important for the site health. We have to be careful while using ZMI so that no action is performed that would cause any malfunction to the site, such as renaming or deleting it. Even adding new content from ZMI is discouraged because Plone will, at best, ignore it.

Following is the ZMI view of a Plone site:

The screenshot displays the ZMI interface for a Plone site named 'intranet'. The left pane shows a tree view of the site's structure, including folders like 'Members', 'acl_users', 'archetype_tool', 'events', 'it-staff', 'mimetypes_registry', 'news', 'portal_actions', 'portal_catalog', 'portal_controlpanel', 'portal_membership', 'portal_metadata', 'portal_modifier', 'portal_properties', 'portal_quickinstaller', 'portal_referencefactories', 'portal_setup', 'portal_skins', 'portal_transforms', 'portal_types', 'portal_view_customizations', 'portal_workflow', 'reference_catalog', and 'uid_catalog'. The right pane shows a list of objects with columns for 'Type Name', 'Size', and 'Last Modified'. The objects are sorted by name. The 'Contents' tab is selected at the top of the right pane.

Type Name	Size	Last Modified
▶▶ HTTPCache		2009-10-13 19:55
MailHost		2009-11-07 18:39
Members (Users)	1 Kb	2009-10-13 19:55
▶▶ RAMCache		2009-10-13 19:55
▶▶ ResourceRegistryCache (Cache for saved ResourceRegistry files)		2009-10-13 19:55
acl_users		2009-10-13 19:55
archetype_tool		2009-10-13 19:55
caching_policy_manager		2009-10-13 19:55
content_type_registry		2009-10-13 19:55
error_log		2009-10-13 19:55
events (Events)	1 Kb	2009-10-13 19:55
front-page (Welcome to Plone)	1 Kb	2009-10-13 20:02
it-staff (IT Staff)	1 Kb	2009-10-18 20:00
kupu_library_tool (Kupu visual editor)		2009-10-13 19:55
mimetypes_registry (MIME types recognized by Plone)		2009-10-13 19:55
news (News)	1 Kb	2009-10-13 19:55
plone_utils (Various utility methods)		2009-10-13 19:55
portal_actionicons (Associates actions with icons)		2009-10-13 19:55
portal_actions (Contains custom tabs and buttons)		2009-10-13 19:55

As we've already noticed, the left frame of the ZMI screen is reserved for a tree view containing the more important objects inside the current object. We can click on **Refresh** at the bottom of the frame to reload it. The right frame holds information about the current object (in this case, the Plone site). As the Plone site itself is a container object, this view also lists all the objects that it contains. The objects are sorted by name, by default, and we can also customize this view and sort them by date or by content type.

We can access the current object attributes, methods, functionality, and properties via the tabs located at the top of the right frame. By default, we always access the **Contents** tab unless we are accessing a non-folderish object. Other relevant tabs are as follows:

- **Security:** Which we can access to manage the permissions for the current object
- **Properties:** Here object properties and attributes are stored
- **Workflows:** Here we can access the information about the current workflow used by the object.

Next, we will mention the most important objects stored in the Plone site root and give a brief explanation of their purpose and functionality.

Object	Description
Mailhost	Tool that stores the settings and methods used by the mail feature. The mail control panel alter ego.
Members	Special Plone folder where personal user folders are stored. It is a special content type optimized for holding a large number of objects (Large Plone Folder).
acl_users	One of Plone's most important tools. This tool manages system users, groups, and roles. In the next chapter, we will cover it in depth.
error_log	This is Plone's error control panel ZMI view.
portal_actionicons	Stores the records that keep the association of each site action with its corresponding icon. It no longer exists in Plone 4.
portal_actions	This tool stores the Plone site actions, such as site map, print screen, or the content views. If we disable Automatically generate tabs in Navigation configlet, we should configure global navigation in its portal_tabs item.
portal_catalog	Another important tool that performs and stores the content indexing information used for all content searches inside the site. It also has methods to manage and maintain this information.
portal_css	An element of the resource registry tool. It stores which CSS (Cascading Style Sheets) files are used in the site and its load preference, or other attributes as well, such as merging and minification.
portal_factory	Responsible for the creation of content objects, it is the place where content lives when the user is adding the content for the first time, so that it can be excluded from search and navigation.
portal_javascript	An element of the resource registry tool. It stores which JavaScript files are used in the site and its load preference, or other attributes as well.
portal_languages	Stores which languages are available in our site along with its related methods and functionality.
portal_memberdata	Holds the default member preference settings.
portal_membership	Saves user policies.
portal_migration	The portal migration tool is used for performing site version upgrades. Plone 4 includes a special UI for migrations.
portal_properties	A container of properties sheets, objects that store a set of attributes for a particular feature, tool, or product.
portal_quickinstaller	This tool takes care of the management of Plone add-on products. The Add-on Products control panel alter ego.

Object	Description
<code>portal_setup</code>	The tool of one of the more powerful features of Plone: <code>GenericSetup</code> . Provides XML import and export information along with the configuration of our site and the products installed. These XML files are called profiles, which are used to configure our site and the add-on products.
<code>portal_skins</code>	This tool stores UI and skin resources from Plone and add-on products, and determines which ones are used for a given skin. It's also the alter ego for the Themes configlet.
<code>portal_transforms</code>	Handles data conversion between MIME types.
<code>portal_types</code>	Controls the available content types in the portal and its properties.
<code>portal_workflow</code>	This tool holds the Plone workflow engine and stores the information about all workflows installed. From here we can also define new workflows, and manage and assign them to content types.

Installing new add-on products

The most common activity when setting up a Plone site is the installation of third party products and new modules. We can achieve this in two ways—via buildout installing the product egg or installing the product as a legacy Zope 2 product.

As an egg via buildout

To illustrate this we will cover some useful cases, the first will be the installation of a new rich text editor—**TinyMCE**.

Using `zc.buildout` is the best way to accomplish our mission, since it is the easiest way to manage the download and its subsequent installation. There is only one requirement: the product has to be packaged as a Python package. **TinyMCE** has been published as a Python package by its author and uploaded to the Python Package Index (PyPI), so `zc.buildout` can download it from PyPI automatically. Its name is `Products.TinyMCE`, and we need to add this line to the `buildout.cfg` file inside the `buildout` or `instance` section.


```
[buildout]
...
eggs=
...
    Products.TinyMCE
```

Next, we must run our buildout. From the buildout's folder we must execute the following:

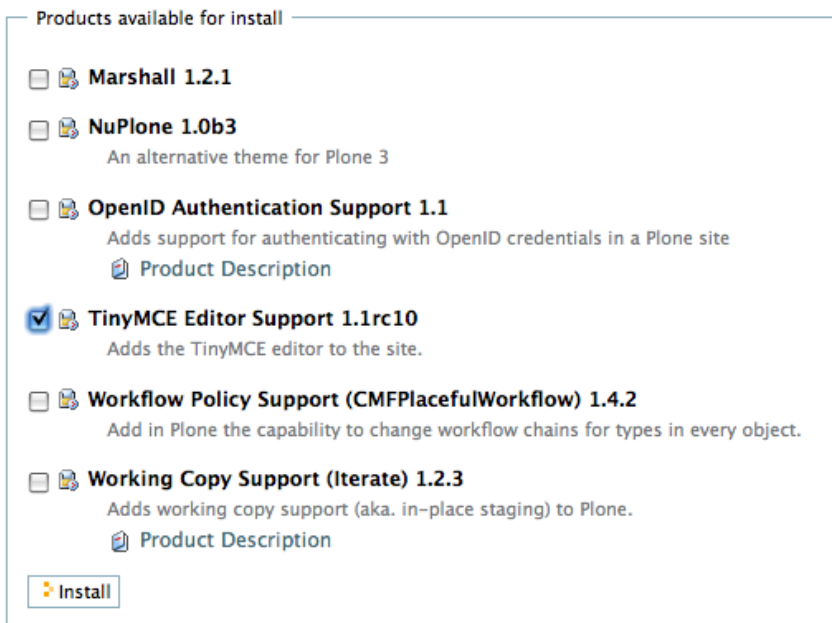
```
$ ./bin/buildout
...
Getting distribution for 'Products.TinyMCE'.
Got Products.TinyMCE 1.1rc10.
...
```

zc.buildout will notice that we want to add this new egg to the installation. It will download the egg from PyPI and add it to the Zope's software path. When it ends, we should stop and start our instance to apply changes:

```
$ ./bin/instance fg
```

 From now on in the book examples, we will always launch the instance in foreground (or debug) mode. To restart our instance, we must stop it by pressing *COMMAND + C* in a MACOSX box, or *Ctrl+C* in a Linux or a Windows box and run it again in foreground mode.

Our product should be shown in the Plone control panel. The **Add-on Products** are ready to be installed, as shown in the following screenshot:



Now that we have installed a new rich text editor, we would choose it as our default text editor for our intranet user from the user menu preferences.



The Products namespace

All the packages with the Products namespace are seen automatically by Zope for historical reasons, so they don't need a **slug** for Zope (often called `zcml slug`) to be able to be aware of them. A slug is a Zope configuration snippet that allows the product to be seen and configured by Zope. We will learn how to create `zcml slugs` next in this chapter.

In case the product we are installing doesn't belong to the Products namespace, then we have to add an additional line to the `buildout` file. For example, let's suppose another practical case: `collective.flowplayer`. This product installs Plone support for this popular open source Flash video player.

We should append these lines to the `buildout.cfg` file in the section instance:

```
[instance]
...
eggs=
...
collective.flowplayer
zcml =
...
collective.flowplayer
```


The highlighted line will tell `zc.buildout` to install a slug in the Zope configuration for allowing the product egg to be seen and configured by Zope. Then, run the buildout again:

```
$ ./bin/buildout
...
Getting distribution for 'collective.flowplayer'.
Got collective.flowplayer 3.0b7.
Getting distribution for 'plone.app.jquerytools'.
Got plone.app.jquerytools 1.0rc1.
...
```

In this case, `collective.flowplayer` has a dependency on the egg `plone.app.jquerytools` resolved by buildout. If we restart Zope, we can see that `collective.flowplayer` is installed:

The screenshot displays the 'Products available for install' section of the Plone Products Manager. It lists several products with checkboxes and 'Install' buttons. The 'Flowplayer 3.0b7' product is checked and has an 'Install' button. Other products include 'Marshall 1.2.1', 'NuPlone 1.0b3' (with a description: 'An alternative theme for Plone 3'), 'OpenID Authentication Support 1.1' (with a description: 'Adds support for authenticating with OpenID credentials in a Plone site' and a 'Product Description' link), 'Plone JQuery Tools Integration 1.0rc1' (with a description: 'Profile for Plone's JQuery Tools resources.'), 'Workflow Policy Support (CMFPlacefulWorkflow) 1.4.2' (with a description: 'Add in Plone the capability to change workflow chains for types in every object.'), and 'Working Copy Support (Iterate) 1.2.3' (with a description: 'Adds working copy support (aka. in-place staging) to Plone.' and a 'Product Description' link). Below this section is the 'Installed products' section, which shows 'TinyMCE Editor Support 1.1rc10' with a description: 'Adds the TinyMCE editor to the site.' and an 'Install log' link. An 'Uninstall' button is visible for this product.

The installation of Flowplayer also triggers the installation of its dependency, Plone's JQuery Tools Integration.

 If we are using the Plone unified installer, we will have to find out the location of our `buildout` directory. It is in a folder called `zinstance` inside the installation folder. If we've used the defaults, it's in `~/Plone/zinstance`. Then follow the same instructions to add a new add-on product as shown in this section.

As a Zope 2 add-on product

If we have downloaded our product as a `tar.gz` from `plone.org` or another source, then we can install it as a Zope 2 add-on product. In our `buildout` folder, go to the `Products` folder. Download the `tar.gz` and uncompress it in this folder:

```
$ cd Products
$ wget http://plone.org/products/mylegacyproduct/releases/1.2/
mylegacyproduct-1.2.tar.gz
$ tar xvfz mylegacyproduct-1.2.tar.gz
```

Then restart the instance and we will have your product available on the Plone control panel.



We can also use the buildout `plone.recipe.distros` recipe as seen in *Chapter 2, Getting Started*.

Summary

In this chapter, we have covered the most important topics about advanced configuration of our Plone intranet's advanced topics (basically about managing our Plone site):

- Plone control panel
- Zope Management Interface (ZMI)
- Installing new products
- Use case: installing other rich text editor
- Use case: installing a non `Products` namespace package

We have also learnt what is behind the Plone scene and showed who is who in the ZMI view. The next chapter will cover user and group management.

5

Managing Users, Groups, Roles, and Permissions

Security, and how to deal with it, is one of the most important topics when building an intranet.

Plone stores users, groups, and security information in the ZODB, but it can also be plugged to almost any existing user database repository, such as LDAP, Active Directory, or SQL-based databases thanks to the **Pluggable Authentication Service (PAS)** included within Plone. This will allow us to use the users stored in that database as Plone users seamlessly.

In this chapter, we will learn the following:

- How security works in Plone
- How to manage local users (basic create/modify/delete operations)
- The mechanisms that Plone makes available to the users to manage their personal data, such as profile data and passwords
- Groups and roles
- The entities of the Plone security

One vision

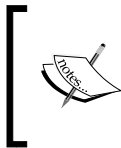
For the first time in the book I want you to do something: *Envision your future intranet*. Take your time; I'm sure you will have a lot of requirements, given by your boss, your client or yourself. Try to order the requirements by their relevance. Take the first item; I bet that it will be a security-related requirement, and in many cases all the first requirements have the same subject – security.

Let's begin with the important issue about security – users and groups. How many users are going to access the intranet we've just envisioned? Depending on the potential number of intranet users, we should choose the backend of the user's database to be used in our intranet.

In case we are going to have few users in our intranet, meaning less than fifty users, then the most basic (and common) use case approach will be to use Plone's (and Zope's) built-in local users stored in the ZODB.

Usually an intranet has an implicitly high volume of user's administration tasks (for example, create, edit roles and permissions, delete, and so on). Dealing with more than fifty users increases the administration overhead exponentially and becomes trickier as this number grows. Although Plone is able to handle any number of users, if we are going to administer more than fifty intranet users, then we might have to start thinking about using an external user repository database.

Another important reason to use an external user database is the existence of an already centralized identity manager. This is very common in mid-sized organizations or companies, where it is usual to find a LDAP-based directory such as Active Directory or OpenLDAP. It's always a good practice to not spawn an independent user repository and to try to plug the Plone PAS into this directory, and instead use the already existent users and groups.



Although this scenario may be very common, it will require some additional Plone skills and we will cover it extensively in *Chapter 13, Deploying our Intranet*. Let's first concentrate on small issues before we begin with the big user use case.

Do we want an intranet where anonymous users could join themselves as members? This is also a possible use case for an intranet, and Plone is very good in handling this special use case as it has a completely automatic join member engine.

Whatever solution we need, we first need to learn about the Plone security entities as they apply to any deployment scenario.

Security entities

First things first, let's call everything by its name and show where they live in our Plone intranet. Users, groups, and roles live inside the Plone PAS object, which is a replacement of the concept known as **user folder** in Zope's jargon.

PAS does a lot more than storing users in the ZODB. It is able to store group objects and manage its users' membership and role objects, and its assignments as well. But the main feature of PAS lies in the word *Pluggable*. We can extend its functionality via plugins to connect it to other user (and group) repositories.

Now, we will introduce the most important topics in Plone (and Zope) security.

Roles

I'm sure that you already know what a user or group is, but it's worth taking a detailed look at roles. Roles are *tags* assigned to our user or group. These tags have a direct security meaning over the rights and permissions of the user or group. Since Plone 3, the names of the default roles are not exposed directly to the user interface for usability reasons. However, we can find their real names in the ZMI and in other Plone UI places, such as the **Users and Groups** configlet in Plone control panel. Following are the roles exposed in the Plone user interface along with their real names:

Role real name	Role UI name	Role default rights
Contributor	Can add	Can add new content to the site
Editor	Can edit	Can edit site content
Reader	Can view	Can read site content
Reviewer	Can view	Can publish site content

However, there are roles that are not exposed to the Plone UI, but are very important:

- **Manager:** Has the maximum rights on the site.
- **Member:** All users by default have this role. It enables the minimum rights every user must have.
- **Authenticated:** A meta-group that includes users authenticated by Zope or another authentication method supported by PAS (for example, OpenID), but not members of the Plone site.
- **Owner:** A special role given automatically to the creator of particular content.
- **Anonymous:** Another special role given to all non-authenticated requests to the site.

The permissions specified by roles are *additive* except for the Manager role which has absolute power. This means that the rights specified by one role are not included in the more permissive role or in any other one. For example, having the Editor role does not imply that one will have the right to read the content too, for this one should have the Reader role.

Global and local roles

There are two ways to assign roles to a user or group: **globally** to the entire site or **locally** to a specific container or content. All the roles assigned via the **Users and Groups** configlet are global, whereas we can assign local roles via the **Sharing** tab located in each Plone content object. The resultant set of roles for a specific user is the sum of both globally and locally specified ones, and in consequence, the resultant rights for that user.

Permissions

Another important entity is the **permission**. The permission is a specific right to perform one security-related thing over an object feature. The permission allows *roles* to have (or not) the right specified by it. For example, the permission **ATContentTypes: Add Folder** controls which roles can add folders in the specified context. Permissions are hard coded inside every object and have parent to child inheritance.



It's very uncommon to directly customize permissions over a specific context. As we will learn shortly in next chapters, it's more usual to modify them by customizing a content workflow as they are overwritten by workflow change. For this reason we won't cover it now. However, if you are curious, you can view the default permissions for any object via the **Security** tab in the ZMI view of every site's object.

Global Zope user accounts

Zope itself has an instance of PAS that contains the default Zope users. We can find it in the **acl_users** object of the ZMI root. Here is where the default Zope administration user is stored, and where we can manage it if needed. This user has full power to manage all objects in Zope. We can add other global users here, such as other Zope administrators, although it is recommended to keep this user folder only for Zope administrator users. More information on this subject and users' administration via ZMI will be available at the end of this chapter.

User self-registration

Plone provides a powerful self-registration engine that enables anonymous users to become registered intranet users using few steps defined by two forms. We can enable it in the **Security** configlet, using the **Enable self-registration** option. Once enabled, all intranet anonymous users will see the **Register** link besides the **Log in** link. This link leads to the user registration form.



All self-registered users will have no roles once created but one: Member. There are only a few things a member user can do, such as manage their own profile data and see other users' profiles, or manage his/her dashboard. Probably we may want to allow the Member role to do more things, such as add new posts in a forum board.

Following are two flavors of the registration form:

- By default, the user is not allowed to choose his password in this step; a URL will be generated and e-mailed to the user. The user must follow the link in the mail to reach a page where the user can change his password and complete the registration process. The user is not active until the whole registration process is complete.
- The other flavor is shown when the option **Let users select their own passwords** is enabled in the **Security** configlet. If enabled, the registration form lets the user specify a password, and if selected, the form can send the password to the user's e-mail address. The user is active immediately.

Managing users and groups

As we have seen in the last chapter, there is a configlet in Plone site setup called **Users and Groups**. From this place, we can access the more important user-related actions and management screens.

The screenshot shows the 'Users Overview' page in a Plone site. At the top, there are three tabs: 'Users' (selected), 'Groups', and 'Settings'. Below the tabs, the page title is 'Users Overview'. There is a link 'Up to Site Setup'. The main text explains that clicking a user's name leads to their details, and an envelope icon is used to email users. It also notes that roles listed apply directly to the user. Below this is a button labeled 'Add New User'. At the bottom, there is a search bar with the text 'User Search:' followed by an input field, a 'Search' button with a magnifying glass icon, and a 'Show all' button with a magnifying glass icon. Below the search bar is a prompt: 'Enter a username to search for, or click 'Show All''.

This configlet provides a convenient user interface for managing users and groups in an easy way.



We can also manage users, groups, and roles directly from the ZMI. Although it is much harder to use UI, in some cases it's more useful for site administrators. If we use an external user database (for example, LDAP) then, probably, we would need to access ZMI for a more quick and accurate control. We are not encouraging the use of ZMI above the Plone UI configlet, but are showing the possible ways to manage users, groups, and roles in Plone. We are going to cover security ZMI management at the end of this chapter.

The user registration form

The **Users** tab holds the control for creating a new user, and for searching and managing site users. The registration form (with its two possible flavors) is the same for self-registration as well as for creating a new user manually by an administrator. Now, let's create a new intranet user. Click on the **Add New User** button. The user registration form is as follows:

Registration Form

Full Name
Enter full name, e.g. John Smith.

User Name ■
Enter a user name, usually something like 'jsmith'. No spaces or special characters. Usernames and passwords are case sensitive, make sure the caps lock key is not enabled. This is the name used to log in.

E-mail ■
Enter an email address. This is necessary in case the password is lost. We respect your privacy, and will not give the address away to any third parties or expose it anywhere.

A URL will be generated and e-mailed to you; follow the link to reach a page where you can change your password and complete the registration process.

Two fields are required: **User Name** and **E-mail**. The other field, **Full Name**, is optional and is used as the display name for identifying the user in the intranet.

Following is an example of the mail the user receives:

```
Welcome Victor Fernandez de Alba,  
your user account has been created. Please activate it by visiting  
    http://localhost:8080/intranet/passwordreset/8e70fbff7161534422b979b2ca3c913b?userid=victor  
  
You must activate your account within 168 hours, so before Dec 08, 2009 09:03 PM  
With kind regards,  
--  
Site Administrator
```

To complete registration, the user must click on the URL sent along with the mail within a week (168 hours). In this final step, the user is prompted to choose his password. Once done, the user becomes active and can be authenticated in our intranet.



Set your password

Please fill out the form below to set your password.

New Password

My user name is
Enter your user name for verification.
victor


New password
Enter your new password. Minimum 5 characters.

Confirm password
Re-enter the password. Make sure the passwords are identical.

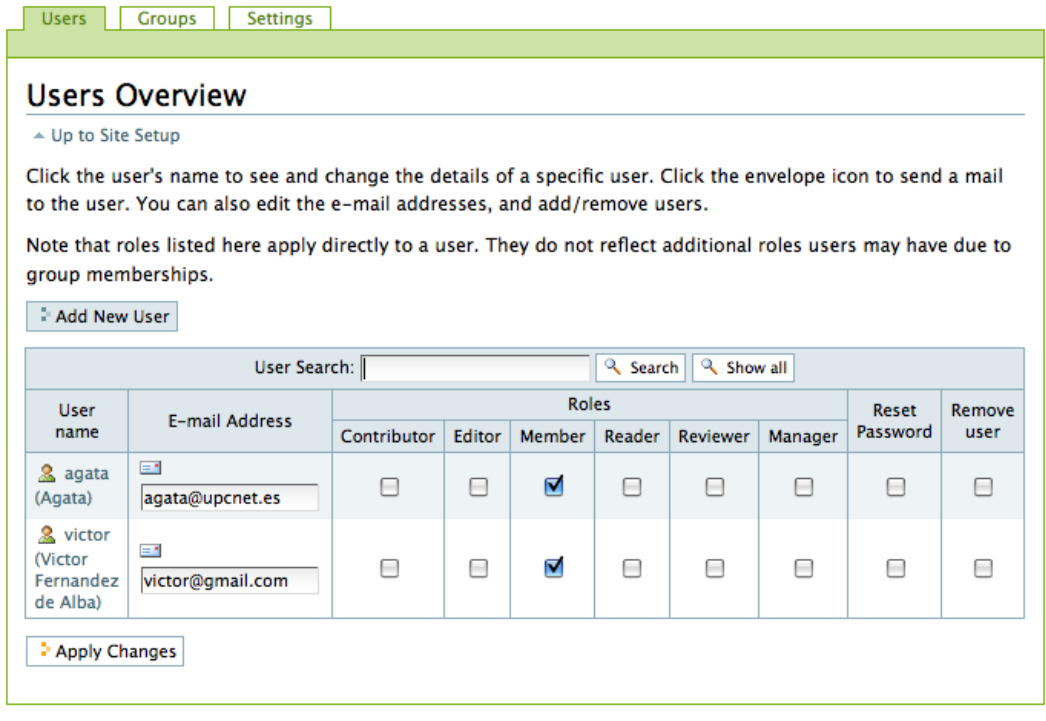
Usernames and passwords have some restrictions: Usernames must contain only alphanumeric characters and no spaces, whereas passwords must have at least five characters.

Managing users

We can manage the users that are already created by searching them using the search controls in the **Users** tab configlet, or simply by clicking on **Show all**, to view all registered users.

 Notice that now, if we click on the **Show all** button, the Zope administrator user will not be seen. Remember that this user is stored in a different user folder, and here we can only manage the users inside the current Plone instance.

Following is the user's overview form once we've clicked on the **Show all** button:



Users Overview





▲ Up to Site Setup

Click the user's name to see and change the details of a specific user. Click the envelope icon to send a mail to the user. You can also edit the e-mail addresses, and add/remove users.

Note that roles listed here apply directly to a user. They do not reflect additional roles users may have due to group memberships.

[Add New User](#)

User Search: [Search](#) [Show all](#)

User name	E-mail Address	Roles						Reset Password	Remove user
		Contributor	Editor	Member	Reader	Reviewer	Manager		
 agata (Agata)	 <input type="text" value="agata@upcnet.es"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 victor (Victor Fernandez de Alba)	 <input type="text" value="victor@gmail.com"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

[Apply Changes](#)

We can see there are two users created at the moment in the screenshot example. Both have the role **Member**, and we can easily assign new roles to them by checking the desired role and clicking on **Apply Changes**. We can reset the user password, and a new password will be sent to the specified e-mail address. It's also possible to modify the user's e-mail address. To remove a user, select the **Remove user** control and apply changes. We can send a mail to the user by clicking on the envelope icon as well.

All the roles given here are **global roles**, so they are valid unconditionally throughout the intranet.

Managing groups

In the **Groups** tab, we can find a similar functionality to the **User** tab. We can either add a new group or manage them. Let's add a new group, for example, the IT staff department group. The create group form is shown with four fields. The **Name** of the group is required whereas the **Title**, **Description**, and **E-mail** are optional.

Assigning a role to a group is as easy as in the user use case. The role is assigned to all the members of that group. Notice that a special group exists – the Authenticated Users. It's a virtual group that includes all authenticated users in our site. Remember its existence, because it's very handy in some security situations.

If we want to add new members to a particular group we must follow the link with the name of the group. A form with three additional tabs is shown in the next screenshot. The first tab is used to add new members to the selected group:

Group Members | Group Properties | Group Portlets

Members of the ITStaff group

[▲ Up to Groups Overview](#)

You can add or remove groups and users from this particular group here. Note that this doesn't actually delete the group or user, it is only removed from this group.

Current group members

There is no group or user attached to this group.

Search for new group members

Quick search:

<input type="checkbox"/>	Group/User name
<input type="checkbox"/>	agata
<input type="checkbox"/>	victor

The **Group Properties** tab form allows us to modify the group attributes (for example, Title, Description, and so on). The **Group Portlets** allows us to set portlets for all the members of the group. Note that group portlets are normally rendered below context portlets. It's a nice feature if we want to show a specific portlet or set of portlets to a list of users and hide them from the rest.

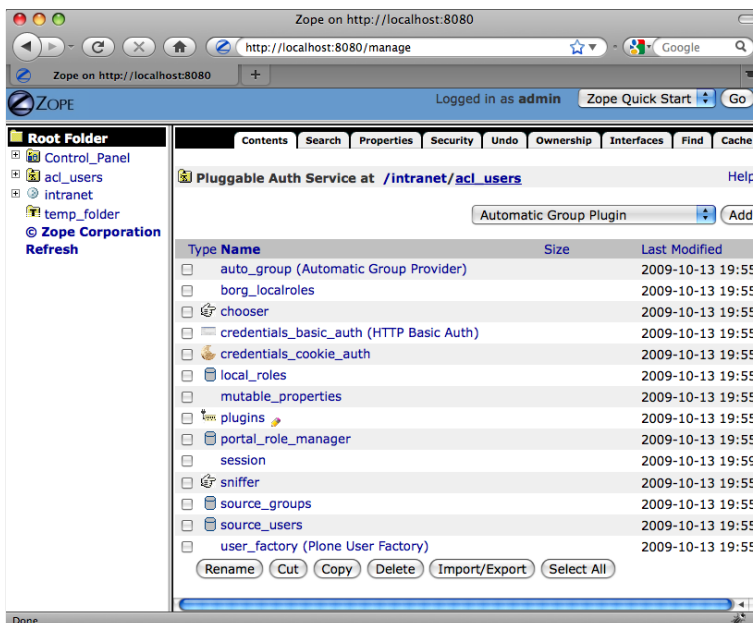
Recovering user password

Plone also provides a way to recover a password in case the user forgets or loses it. This feature is available from the login form or the login portlet. This leads to a form where the user is prompted for their username. For security reasons, the password is not sent to the user, instead, a password reset process is triggered, and an e-mail with a special URL is sent to the e-mail address of the user.

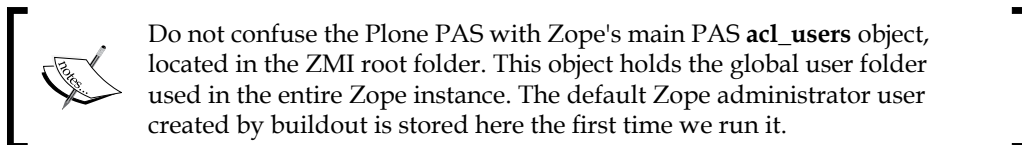
More control: managing ZMI

If we need more control, we can access the Plone PAS ZMI object directly. There's a probability that we may find it more useful and quicker than the Plone configlet. Although it has a hard UI, we will cover it because it's another tool available for intranet managers and is worth having a look.

The Plone PAS object is called **acl_users** and is located in the root of the Plone site in the ZMI. This is what it looks like:



One of the important features of Plone PAS is that it contains a standard user folder, and in addition, we can add plugins to this user folder to extend it. We can extend it with users and groups existent in another user repository such as LDAP or Active Directory.



The Plone PAS itself is composed of the following main objects:

- **source_users:** A user objects manager
- **source_groups:** A group objects manager
- **portal_role_manager:** A role objects manager
- **plugins:** Stores all the available plugins, lists them, and shows which are active

The other objects that Plone PAS contains are plugins in use or other support objects to help Plone PAS work correctly.

Administering users via ZMI

We can create a user in a few steps using the `source_users` object. We can easily add a user by clicking on the **(Add a user)** link and then completing the form. All fields are required: **User ID**, **Login Name**, and **Password**. Once created, we can manage all users with the simple controls provided. The **Password** link will lead us to a reset password form for the selected user. We can also remove the selected user from this form. Following is the `source_users` form for the user created previously:

User Manager at /intranet/acl_users/source_users

Current Users (Add a user)

	User ID	Login Name
<input type="checkbox"/>	victor	Password victor

[Remove Users](#)

Administering groups via ZMI

The `source_groups` object works very similarly to the previous one, but additionally manages the user membership to the site groups. We can create a new group by clicking on the **(Add a group)** link, and completing the form. **Group ID** is required, while **Title** and **Description** are optional. In the example, we have created a new group called **HResources**, and we plan to add all the human resources users as members.



We can add a member to the group by clicking on the ? character. The opened form will allow us to search users in the user folder and assign them to the current group using the controls provided.

Administering roles via ZMI

The `portal_role_manager` object is the place where we can manage roles and role assignments for users and groups. The interface is the same for users and groups, so we can basically add a new role by following the **(Add a role)** link, and manage role assignments using the form located in the ? link.



We will never need to add an additional role unless we want to customize security and workflows heavily.

Following is the `portal_role_manager` view:



The screenshot shows the 'Group Aware Role Manager' interface at the URL `/intranet/acl_users/portal_role_manager`. The main heading is 'Current Roles (Add a role)'. Below this is a table with three columns: 'Role', 'Description', and 'Assignments'. Each row represents a role with a checkbox on the left. The roles listed are Contributor, Editor, Manager, Member, Owner, Reader, and Reviewer. The 'Assignments' column shows various users or groups assigned to each role, such as '? Administrators' for Manager and '? victor' for Member. A 'Remove Roles' button is located at the bottom of the table.

Role	Description	Assignments
<input type="checkbox"/> Contributor		?
<input type="checkbox"/> Editor		?
<input type="checkbox"/> Manager		? Administrators
<input type="checkbox"/> Member		? victor
<input type="checkbox"/> Owner		?
<input type="checkbox"/> Reader		?
<input type="checkbox"/> Reviewer		? Reviewers

Remove Roles

The sharing tab

All we have learnt so far applies to global security. When we assign a role to a user or group in the **Users and Groups** configlet or in the ZMI `acl_users`, these assignments are global to the entire Plone intranet. This means that the user or user members will always have these roles, unconditionally, throughout the site.

So this brings us to the first most common security-related use case: As an administrator, I want to have more control over given roles than I have with a unique set of global roles. The mighty **Sharing** tab (and local roles) come to the rescue!

All content in our site (including the Plone site object itself) can store local role assignments for users and groups. These roles are added to the global ones and results in the total role set for a user or group. We can manage the local roles of every object using the **Sharing** tab. Click on it to access the sharing form shown in the next screenshot:

The screenshot shows the 'Sharing' tab of a Plone interface for an object named 'IT Staff'. The interface includes a search field for users or groups, a table of permissions, and a checkbox for inheriting permissions from higher levels.

Sharing for "IT Staff"

You can control who can view and edit your item using the list below.

Search for user or group

User/Group	Can add	Can edit	Can view	Can review
Logged-in users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Inherit permissions from higher levels
By default, permissions from the container of this item are inherited. If you disable this, only the explicitly defined sharing permissions will be valid. In the overview, the symbol indicates an inherited value. Similarly, the symbol indicates a global role, which is managed by the site administrator.

The sharing form's highlight is its simplicity: a field for searching the desired user or group to assign new local roles, and a table for selecting the role. As we said before, for usability reasons, the real names of the roles are not exposed in the Plone UI. Only a more meaningful description is given. As can be seen in the previous screenshot, the virtual group **Logged-in users** is shown in case we need it, but with no local roles set by default.

Local role inheritance

An important feature of local roles is their inheritance. Local roles inherit from parent to child objects. However, this behavior can be overridden by an existent control in the sharing form – **Inherit permissions from higher levels**. If we disable this, only explicitly defined permissions will be applied to this object and in consequence to all its children objects.

This leads us to another common security use case: As a user, I want to be able to set up a folder where only some other person(s) and me have read and edit rights. This is the moment when the ability to cut inherited permissions is very useful.

Of course, we can always re-enable inheritance for a container to regain the rights from the parent object.

Summary

At this moment, we already know the tools Plone provides to deal with security.

In this chapter, we have learnt about the following key topics:

- The Plone Pluggable Authentication Service
- Users
- Groups
- Roles
- Zope permissions
- The Sharing tab

These concepts, along with workflows, is all we need to know to manage security effectively in a Plone based intranet. In the next chapter, we will cover workflows in depth. By the end of the next chapter, we will be ready to put in practice all these learned topics.

6

Managing Workflows

Now, we are facing our last step to learn and understand Plone security completely. Workflows are an important security tool and also a powerful content life cycle engine. Plone provides a set of preconfigured workflows covering the most common life cycle use cases. The workflow engine allows us to create a customized workflow and modify the existing ones too.

Workflows provide the following features:

- Defining a state per content object
- For that state, defining the object's current security
- Defining possible transitions to other states and the events responsible for these transitions
- Defining approval points for each state

Unfortunately, Plone doesn't provide any graphical user interface for managing workflows, and, hence we will have to use ZMI or rely on some third-party tools. We will cover some of these tools, such as workflow diagram drawers, that will help us see graphically the state diagram of our customized workflow.

Some time ago, Martin Aspeli wrote about a very useful product for managing workflows – `collective.wtf`. It features import/export to/from a CSV file and some other interesting workflow-related tools. Carlos de la Guardia made another useful product called `collective.workflowed`. It provides a graphical workflow editor with drag and drop capabilities.

This chapter will cover the following topics:

- Elements of a workflow
- Out-of-the-box workflows state diagrams
- Modifying an existing workflow
- Must have tools: `DCWorkflowGraph`, `collective.wtf`, and `collective.workflowed`
- Workflows best practices

Workflow entities

Typically, a workflow is composed of two major entities — **states** and **transitions**. However, there are other entities responsible for the other, not-so-well-known, capabilities of the Plone workflow engine.

States

As we all know, all content types in Plone are eligible to have a workflow assigned, and in consequence, to have a life cycle and a current state. However, a content type may not have a workflow assigned. This only means that they hold no state information and inherit permissions from their parent. By default, images and files are not assigned to any state.

Basically, a content state determines the security permissions for any content object in that state. When the workflow enters a state, then it is given a chance to update permissions on the object.

Transitions

All workflows start in a particular state called the **initial state** and then move on to other states via transitions. They are actions that define the destination state and other attributes such as transition guards, scripts to be executed (before and after changing state), and the way the transition is displayed in the action box in Plone UI.

Each state defines which exit transitions are available for that state, and the transition that controls the destination state after the transition is complete. This destination state is unique. Transitions are user-triggered by default, but may be automatic. Automatic transitions only happen at the same time as other transitions if the assigned guard (see the next section) evaluates to true. It is also possible to define the transition to stay in the current state. This can be useful if we want to execute one action over the object and take record of it, but not trigger a state change (and in consequence permissions) for that object.

Guards

Usually, a change of state cannot be triggered by anyone, and for that reason there exists an attribute called **transition guards** for all transitions. This attribute evaluates true or false against the current user security rights and can be any of these:

- A particular permission, for example `Review portal content`
- A role or list of roles
- A group or list of groups
- A **Template Attribute Language Expression Syntax (TALES)** expression

Guard conditions ensure that only those users with the required permission, role, group membership, or other criteria, can move the object to the new state via the TALES expression.



The TALES describes expressions that may be used to supply **Template Attribute Language (TAL)** and **Macro Expansion Template Attribute Language (METAL)** with data. It's the standard attribute language used in Zope to create dynamic templates. To learn more about them visit http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/AppendixC.stx.

An automatic transition triggers immediately, following another transition, provided its guard conditions pass.

Permissions

Each workflow manages a set of permissions to be applied each time a state change is triggered. Plone assigns a default set of permissions, based on the default installed workflow. These permissions will vary depending on the type of workflow we assign to our content, but by default they are some core content management permissions such as **View**, **Modify portal content**, and so on. However, we can also assign more permissions to be applied by that state. We must choose which permissions the workflow will manage, and then define them individually for each state.

Assigning local roles to groups

This is like assigning roles to groups on Plone's **Sharing** tab; but the mapping of roles to groups happens on each state change, much like the mapping of roles to permissions. Combining these with the role to permission mapping is very powerful.

Scripts

Attached to any transition, it is possible to define scripts to be executed before and after the state change occurs. These are created through the Web as External Methods. We can customize these scripts to do certain actions over content or other kinds of things, for example, sending mail.

ZMI workflow management

All workflow managing tasks have to be done using ZMI, except assigning already existing workflows to content types and actions that can be done through Plone's **Types** configlet, located in the site setup.

The `portal_workflow` is the ZMI object that contains all workflow related objects and tools. We can access it from the ZMI view of our Plone site. The following screenshot is its default view:

The screenshot shows the 'Plone Workflow Tool' interface. At the top, there are navigation tabs: Workflows, Overview, Contents, View, Properties, Security, Undo, Ownership, Interfaces, and Find. The main heading is 'Workflows by type'. Below this, a list of workflow types is shown on the left, and their corresponding workflow objects are listed on the right. The workflow objects are mostly '(Default)', but the '(Default)' object is associated with 'simple_publication_workflow'.

ATBooleanCriterion (Boolean Criterion)	
ATCurrentAuthorCriterion (Current Author Criterion)	
ATDateCriteria (Friendly Date Criteria)	
ATDateRangeCriterion (Date Range Criterion)	
ATListCriterion (List Criterion)	
ATPathCriterion (Path Criterion)	
ATPortalTypeCriterion (Portal Types Criterion)	
ATReferenceCriterion (Reference Criterion)	
ATRelativePathCriterion (Relative Path Criterion)	
ATSelectionCriterion (Selection Criterion)	
ATSimpleIntCriterion (Simple Int Criterion)	
ATSimpleStringCriterion (Simple String Criterion)	
ATSortCriterion (Sort Criterion)	
ChangeSet	(Default)
Discussion Item	
Document (Page)	(Default)
Event	(Default)
Favorite	(Default)
File	
Folder	(Default)
Image	
Large Plone Folder (Large Folder)	(Default)
Link	(Default)
News Item	(Default)
Plone Site	
TempFolder	(Default)
Topic (Collection)	(Default)
(Default)	simple_publication_workflow

Click the button below to update the security settings of all workflow-aware objects in this portal.

[Update security settings](#)

We will first see the workflow view in this object. In this view, we can find the workflow assignment to all content types available in the site. This information is the ZMI reflection of its Plone UI configlet alter ego: **Types**. We can change the default assigned workflow of a particular type either from here or from the configlet.

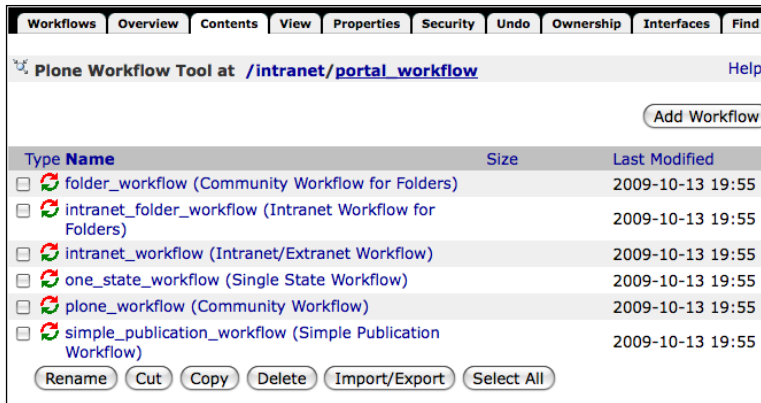
We can perform the change by modifying the name of the workflow in the fields next to any content type name. **(Default)** means the content type has assigned the default workflow defined at the bottom of the view. It is assigned by default to the **simple_publication_workflow**. As we can see, one must specify the ID of the object corresponding to the desired workflow to be assigned. We can find out the IDs of the created workflows available in the **portal_workflow** tool in the **Contents** tab.

If we change one or more workflows from this view, then we must execute the action triggered by the button located at the bottom of the view, **Update security settings**. It causes an update of all the existent objects of that specific content type to the new workflow specified. If we don't execute it, then the change will be applied only to those objects created from the moment we changed the workflow settings for that kind of content type.

However, it is always recommended to perform this action using the **Types** configlet. It allows us to specify additional actions that we can perform on the affected objects. The most important one is the mapping of states from the original workflow to the destination one. This is important because the most probable scenario is that the original workflow doesn't have exactly the same state as the destination one. The configlet allows us to determine the destination state for all the original states. Because of this, no object is left with an undetermined workflow. Depending on the number of objects we may already have on our site, the update to security settings may last a few minutes.

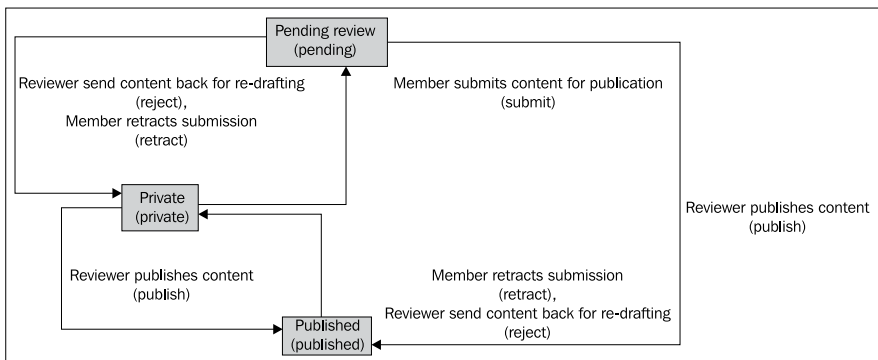
Out-of-the-box workflows

Now it's time to show the predefined workflows that Plone provides. These workflows are intended to cover the most usual use cases. They are extensible and customizable, like any other existent Plone workflow. We can find them in the **Contents** tab of the **portal_workflow** object.



Simple publication workflow

Simple publication workflow is the default workflow assigned to all content types. Following is the state diagram of a simple publication workflow:

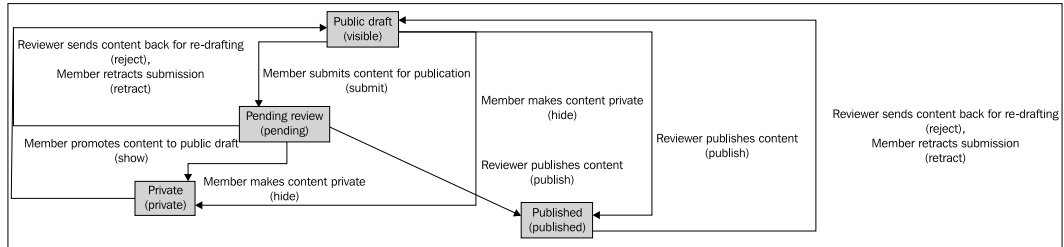


Content types assigned to this workflow will start out as private, and can either be submitted for review, or published directly. The creator of a content item can edit the item even after it is published. Following is a more detailed overview of each state:

- **Private:** Content can't be accessed by anonymous users, and only editors, managers, and owners can modify it.
- **Pending:** Content can't be accessed anonymously. Contributors, readers, and editors can access content, and only reviewers and managers can edit and publish the content.
- **Published:** Content can be accessed by everyone, but only editors, managers, and owners can modify it.

Community workflow

Community workflow is also known as "Plone workflow", as it was the default workflow for Plone 3 previously. It allows users to create content that is immediately, publicly accessible. Following is the state diagram of community workflow:

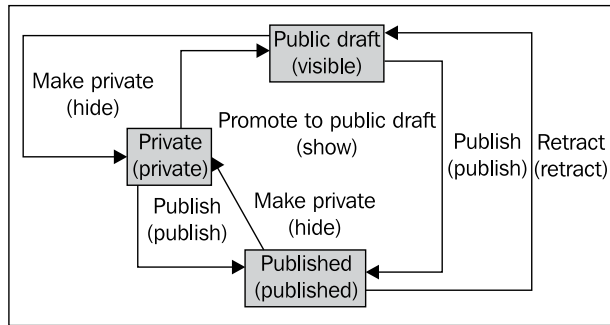


Content types assigned to this workflow will start out as public draft and will be visible to anonymous users. The content's creator, or a user with the Manager role, can submit content for publication. Owners and editors can make content private, but this workflow indicates that it's not published to anonymous users. Reviewers can publish or reject content, whereas content owners can retract their submissions. While the content is awaiting review, it is readable by anyone. If content is published, only a manager can retract it. Following is the detailed state overview:

- **Draft:** Anyone can access content in this state. Editors, managers, and owners can modify it.
- **Private:** Anonymous access is forbidden. Contributors and readers can access it, but only editors, managers, and owners can edit the content.
- **Pending:** Anonymous access is allowed. Only managers and reviewers can modify the content.
- **Published:** Content is accessible anonymously, but only managers can edit it.

Community workflow for folders

Community workflow for folders is normally used in conjunction with the community workflow and assigned to the folder content type. It has no pending state, as it is not needed for a folder. It allows the owner to publish the folder without approval. Following is the state diagram for this workflow:



Like in community workflow, content types assigned to this workflow will start out as public draft and will be visible to anonymous users. In more detail:

- **Draft:** Anonymous users have access. Contributors, members, and readers can list folder contents, and editors, managers, and owners can modify it.
- **Private:** Content can't be accessed anonymously. Portal users can access content and list its contents, while editors, managers, and owners can modify it.
- **Published:** Anyone can access the content and list its contents, but only editors, managers, and owners can change it.

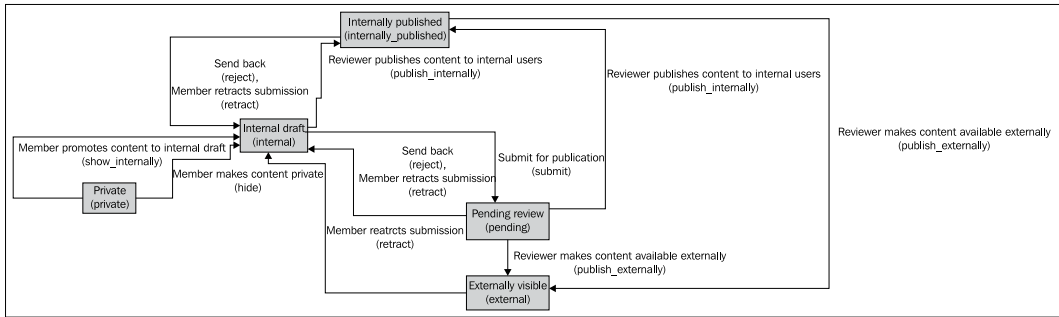
One state workflow

One state workflow is essentially a workflow with no transitions, and it only has a unique state—published. The published state is chosen for this workflow because there are portlets and applications that expect this state to exist in order to work properly. Content in this state can be accessed anonymously, and editors, managers, and owners can modify it.



Intranet workflow

In intranet workflow content is accessible only if we are logged in. Basic states are: Internal draft, Pending review, Internally published, and Private. An additional state is also available: Externally visible. It allows us to make selected content available to people outside the intranet.

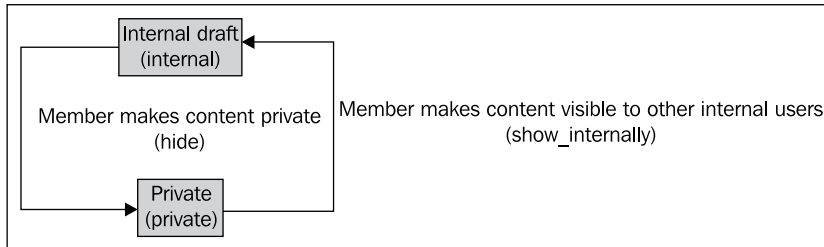


Following is a detailed overview of the states of this workflow:

- **Internal draft:** Only editors, managers, and owners can modify the content, while it can be accessed by any portal member.
- **Private:** Only editors, managers, and owners can modify the content, while contributors and readers can access it.
- **Internally published:** Any portal member can access it, but only managers can modify it.
- **Pending review:** Any portal member can access it, but only managers and reviewers can modify it.
- **Externally visible:** It can be accessed anonymously and modified only by managers.

Intranet workflows for folders

This workflow is used for complementing the intranet/extranet workflow and it's typically assigned to folderish types. They only have two states: Private and Internal draft.



This simple workflow for intranet folders has the following state overview:

- **Private:** Contributors and readers can access contents and list folder contents, whereas editors, managers, and owners can modify it.
- **Internal draft:** Portal members can access and list its contents, and editors, managers, and owners can modify it.

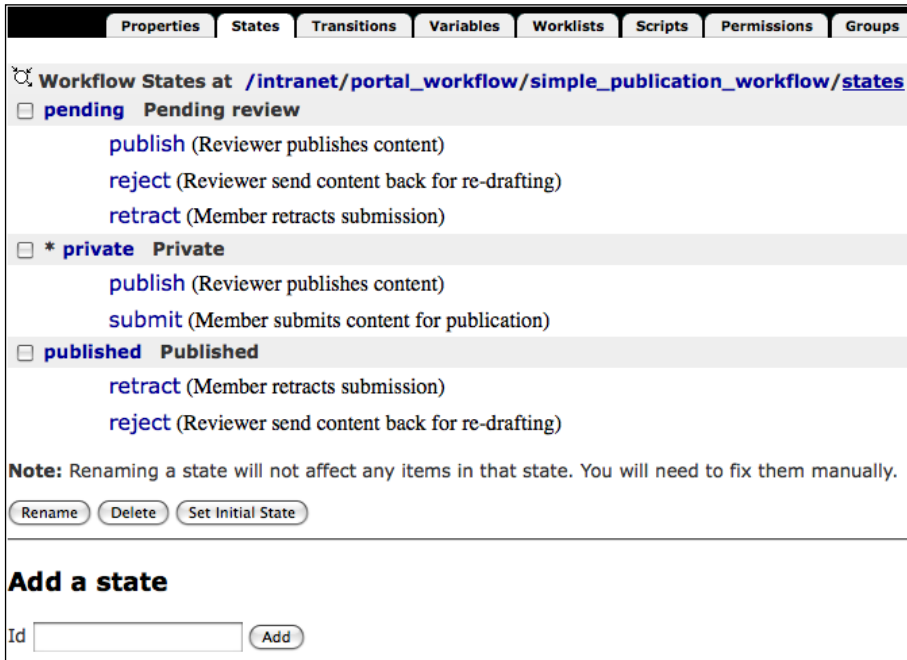
Workflow diving

Let's take a deep breath and dive into the contents of a workflow. We've already shown its most important entities and now is the time to show them in action. Let's take the standard Plone workflow. We will use the simple publication workflow as an example for this section. Just click on it in the **Contents** tab. The default view of a workflow object is the **Properties** tab. It holds some interesting attributes of the object, such as:

- Title
- Description
- "Manager" role bypasses guards. If checked, then all users with the Manager role will be able to access all available transitions, regardless of the guards defined.

States

The **States** tab shows all defined states for the current workflow and the controls needed to manage them and create new ones. Following are the states of the simple publication workflow:



We can see that, for each state, there are some associated possible transitions. Later, we can access the properties of these transitions by clicking on them. At the bottom of the view we can see the controls for managing states. We can rename, delete, and set the default initial state for all content objects assigned with the current object. A * besides the ID of the state means that it is the initial state for this workflow. To add a new state, a text field is provided.

We can access the attributes of any state by clicking on its ID. By doing this, we are accessing a new view where we can find the following, distributed across different tabs:

- **ID:** Unique and assigned to a state on creation.
- **Title:** The display name of the current state. It's i18n aware.
- **Description:** Only for information purposes.

- **Possible transitions:** A list of possible transitions from a state to another state.
- **Permissions:** The CMF permission set to be applied when the content object is assigned to a state.
- **Groups:** To be added in the list of local roles of the content object.
- **Variables:** The variables to be added.

We can probably find the most important setting of a state under the **Permissions** tab, which is the managed permissions to be applied on content state change. It will help us configure these managed permissions:

Properties		Permissions		Groups		Variables				
Workflow State at /intranet/portal_workflow/simple_publication_workflow/states/published										
When objects are in this state they will take on the role to permission mappings defined below. Only the permissions managed by this workflow are shown.										
Permission	Roles	Anonymous	Authenticated	Contributor	Editor	Manager	Member	Owner	Reader	Reviewer
Acquire permission settings?										
<input type="checkbox"/>	Access contents information	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	Change portal events	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	List folder contents	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	Modify portal content	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	View	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="button" value="Save Changes"/>										

Remember that the role of permission mappings for an object in this workflow depends on its state. The default permissions to be managed by this workflow are the most basic CMF permissions:

- Access contents information
- Change portal events
- List folder contents
- Modify portal content
- View

For each permission in the list, we may define which roles are allowed. These settings will be applied to the already existing permissions in the content object, overwriting them at the moment of state change.



All changes that we may perform in a state does not get applied to content objects immediately; they only apply to all content from the moment we made the change. If we want to apply the change to all existing content, we must execute an update workflow action, located in the `portal_workflow` tool object view. This action is the **Update security settings** button that triggers a workflow update to all the objects of the site.

It is possible to add more managed permissions in the **permissions managed by this workflow** link, located in the description of this view. The link leads us to another view where we may set more permissions that will be later managed by this workflow. We can either select more or remove the existing ones.

Transitions

We can access the attributes of any transition either from the states main view or from the **Transitions** tab.

Properties	States	Transitions	Variables	Worklists	Scripts	Permissions	Groups	
<p>Workflow Transitions at /intranet/portal_workflow/simple_publication_workflow/states/transitions</p>								
<input type="checkbox"/>	publish	Reviewer publishes content	Destination state: published Trigger: User action Requires permission: Review portal content Adds to actions box: Publish					
<input type="checkbox"/>	reject	Reviewer send content back for re-drafting	Destination state: private Trigger: User action Requires permission: Review portal content Adds to actions box: Send back					
<input type="checkbox"/>	retract	Member retracts submission	Destination state: private Trigger: User action Requires permission: Request review Adds to actions box: Retract					
<input type="checkbox"/>	submit	Member submits content for publication	Destination state: pending Trigger: User action Requires permission: Request review Adds to actions box: Submit for publication					
<p>Note: Renaming a transition will not automatically update all items in the workflow affected by it. You will need to fix them manually.</p>								
<p style="text-align: left;"> <input type="button" value="Rename"/> <input type="button" value="Delete"/> </p>								
<p>Add a transition</p>								
<p>Id <input style="width: 100px;" type="text"/> <input type="button" value="Add"/></p>								

Like in the state's view, we can access an extended transition properties view by clicking on each transition ID. However, this view also shows some of the most important attributes as a resume. Controls for managing and creating new transitions are also available.

A transition has the following properties:

- **ID:** Unique and assigned to the transition on creation.
- **Title:** The display name of the transition and `i18n` aware.
- **Description:** The extended description available as the title attribute of the corresponding link for that transition in the Plone UI and `i18n` aware.
- **Destination:** State.
- **Trigger type:** Can be automatic or initiated by user action.
- **Scripts:** To be executed before and after the transition is completed.
- **Guard:** A set of checks against a permission, a role, a group, or a TALES expression that determines whether a transition should be shown to the current user or not. If it evaluates true, the transition will be available. If not, the user will not be able to trigger the transition.
- **Display in actions box:** These are some attributes related to the Plone UI and determine how to show the transition. They also determine the `i18n` string ID for that transition in the **Name (formatted)** field.

Variables

Every time a user triggers a state change it results in a number of variables being recorded, such as the actor (the user that invoked the transition), the action (the ID of the transition), the date and time, and so on. Each workflow can define any number of variables linked to TALES expressions that are invoked to calculate the current value of the variable at the point of transition. We can find them in the **Variables** tab for every site workflow. From this view, we can modify existing variables, and create new ones.

In addition, the state is exposed as a special type of workflow variable called the **state variable**. Most workflows in Plone use the name `review_state` as the state variable. This is hard coded for the existing workflows and we must set it explicitly in the **State variable name** if we are adding a new workflow from scratch.

Worklists

A worklist is a stored query executed against the database that returns a list of objects that are in a particular state. We can either modify or create new worklists using the **Worklists** tab.

For example, worklists are used in Plone's review portlet. Plone's review portlet shows all current worklists from all installed workflows. The use of worklists means that we can display all the items that are together in all the worklists, which apply to the current user in a single portlet. Most Plone workflows have a single worklist that matches with the `review_state` variable, for example, the pending state.

Scripts

Workflow scripts are usually created through the Web, using a `script` (Python) object or an External Method. We can add one of them via the drop-down box, which is new to the **Add button**, and then link them from any transition.

No workflow and multiple workflow use cases

There are two special situations we haven't discussed yet. The no workflow assigned case is used when we don't need to apply any particular workflow at all to a content type. By default, Plone has two content types that are not assigned to any workflow — images and files. The reason for this is that it's more comfortable to publish a page that contains these kinds of types, because we don't have to publish each image or file contained in the page in order to make it available. These two content types will inherit permissions from their parent.

Adding multiple workflows to a single content type



A content type can be assigned to several workflows. We can list multiple workflows by separating their names with commas. This is called a workflow chain. Multiple workflows can be very useful when we have concurrent processes. Multiple workflows applied in a single chain always co-exist in time. Plone will show all available transitions from all workflows in the current object's chain in the state drop-down box. However, this is an advanced feature and should be used with care. We can face issues, such as variable collisions, if we assign two of the default Plone workflows.

Some useful workflow tools

Although Plone has one of the most powerful workflow engines out there, it lacks the convenience of some sort of graphical management tool. However, there are some tools out there to help us overcome this and other workflow matters.

DCWorkflowGraph

Although `Products.DCWorkflowGraph` is an old, well-known Plone add-on, it continues to be as useful as the first day. It analyzes a particular workflow and generates a graphical state and transition diagram. It's always invaluable to have a graphical representation whenever dealing with a complex workflow, especially while creating or modifying it.

To install it, we must add it to the `buildout.cfg` file in the `eggs` attribute of the `buildout` section:

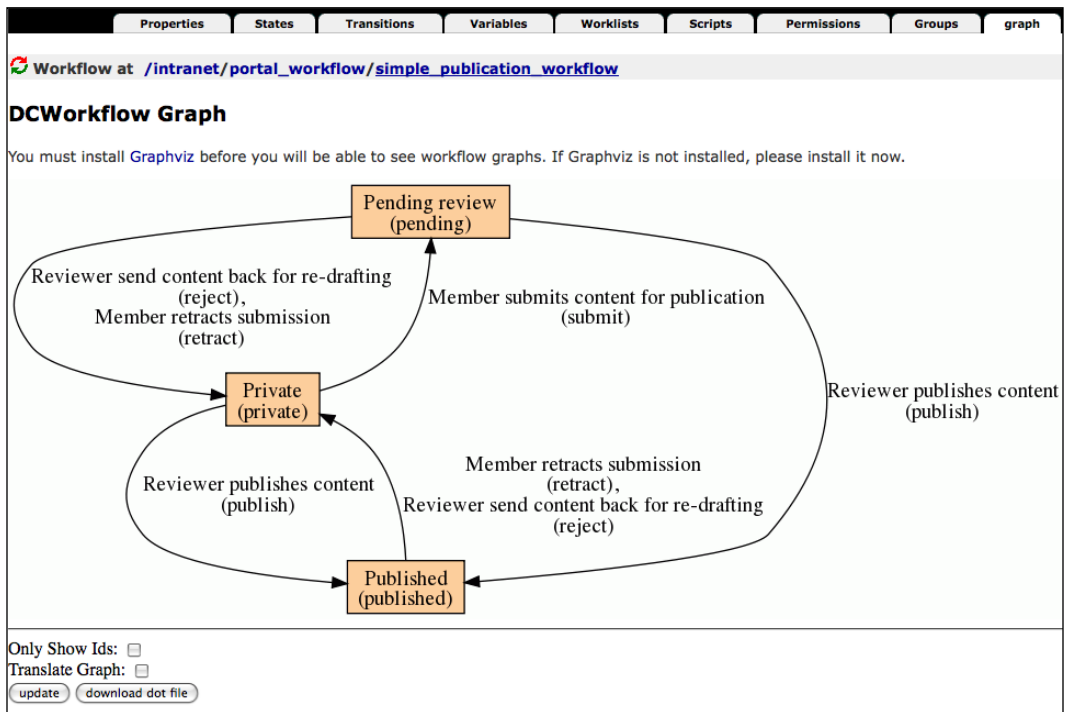
```
[buildout]
...
eggs = Products.DCWorkflowGraph
```

Then, remount buildout:

```
$ ./bin/buildout
$ ./bin/instance fg
```

`DCWorkflowGraph` has one dependency – we must install `GraphViz` (<http://www.graphviz.org/>) in our system. There are install packages for all platforms.

Once installed, we will have a new tab called **graph** available in each workflow object:



We can see the states represented in orange squares, and the respective transitions represented by arrows indicating the direction of the transition.

collective.wtf

This add-on product written by Martin Aspeli is a GenericSetup importer/exporter tool that instead of using XML, uses a CSV file. It also provides a number of debugging aids around workflows. For example, we can easily get a CSV view of a currently installed workflow to sanity-check permissions, and there is a view that runs some heuristics on our installed workflows to check against Plone conventions and best practices.

To install it, we must add it to the `buildout.cfg` file in the `eggs` attribute of the `buildout` section:

```
[buildout]
...
eggs = collective.wtf
```

Also, add it to the `zcml` attribute of the `instance` section:

```
[instance]
...
zcml = collective.wtf
```

Then, remount buildout:

```
$ ./bin/buildout
$ ./bin/instance fg
```

For exporting a CSV file containing the definition of our workflow, type a URL such as the following in the browser:

```
http://localhost:8080/intranet/portal_workflow/my_workflow/@@to-csv
```

To invoke the sanity checker, type a URL such as the following into the browser:

```
http://localhost:8080/intranet/portal_workflow/my_workflow/@@sanity-check
```

In both cases, `intranet` is the name of our Plone site, and `my_workflow` is the ID of the workflow we desire to dump.

Another great tool that this product provides is the ability to view the current roles of a given user in a given context. Type the following URL into the browser when logged in as a manager user:

```
http://localhost:8080/intranet/context/@@display-roles-in-context?user=<user>
```

Again, `intranet` is the name of the Plone instance and `context` could be any object. The `<user>` string should be replaced by the login name/ID of the user we want to fetch roles for. The output will be shown in the browser window in plain text.



We can find more information about `collective.wtf` in the following URL: <http://pypi.python.org/pypi/collective.wtf>.

collective.workflowed

This is another very useful add-on created by Carlos de la Guardia. Essentially, it is a graphical workflow editor for Plone. It is also a wrapper of the features of `collective.wtf` and exposes some workflow editing, such as permissions to the Plone UI. We might find this add-on useful if we missed a graphical utility to manage workflows for Plone.

To install it, we must add it to the `buildout.cfg` file in the `eggs` attribute of the `buildout` section:

```
[buildout]
...
eggs = collective.workflowed
```

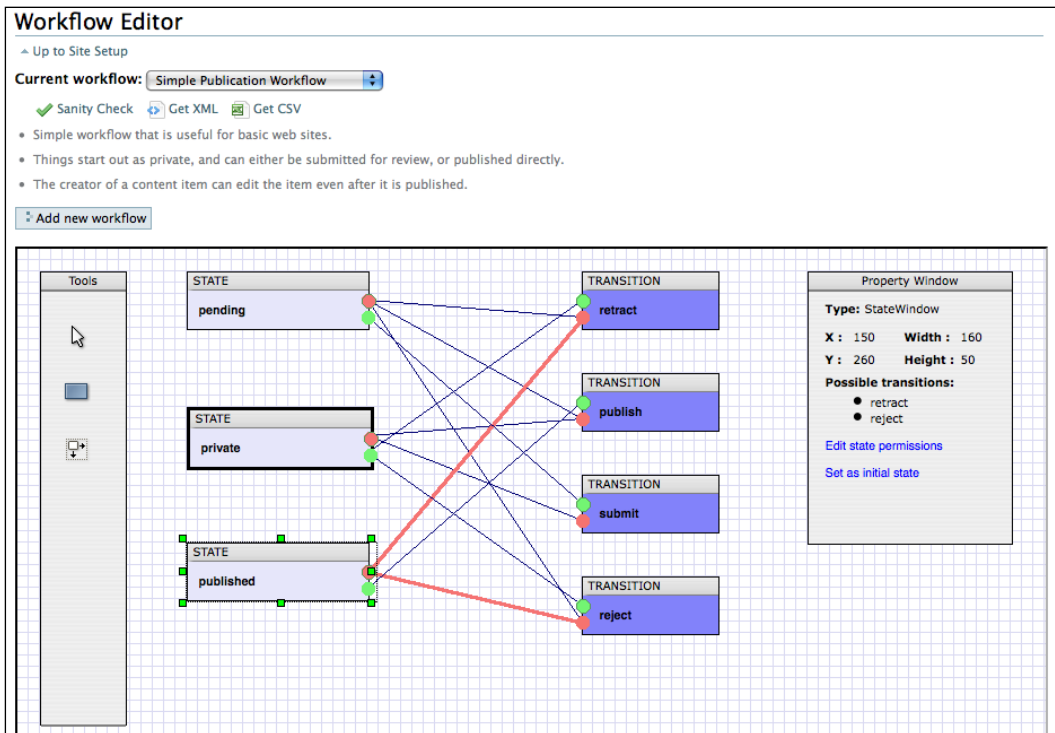
Also, add it to the `zcml` attribute of the `instance` section:

```
[instance]
...
zcml = collective.workflowed
```

Then, remount `buildout`:

```
$ ./bin/buildout
$ ./bin/instance fg
```

We must install the product from the Add-on products configlet. It will install a new application configlet called `Workflow Editor`, as shown in the following screenshot:



The most distinctive part of this product is its active zone with full drag and drop support, based on the powerful Draw2D library from the Openjacob project (<http://draw2d.org/draw2d/>). In this zone, we can see a graphical representation of the state diagram of the workflow. We can modify the current selected workflow by using the provided controls, add new states and transitions, and edit state managed permissions. It's possible to add an entirely new workflow as well.



We can find more information about `collective.workflowed` in the following URL: <http://plone.org/products/collective-workflowed>.

Placeful workflow

Shipped with Plone since 3.0, it is a very useful tool. It allows us to configure an alternative workflow for one or more content types in or below a particular folder, overriding the global workflow setting. This setting is called a workflow policy. To apply a policy, we will have to add a local workflow policy configuration to the desired folder using the **policy** item in the state drop-down menu of any folder.

To install it, we must go to the Add-on Products configlet and install the Workflow Policy Support (CMFPlacefulWorkflow). Once done, we have new configlet available – Workflow Policies. This configlet will allow us to create and configure our workflow policies. They consist of the following components:

- **ID:** A unique name for the policy
- **Description:** For information purposes
- **Default workflow:** The initial workflow type assigned to the content types of the folder where the policy is applied
- **Workflows by type:** Here we can specify a particular workflow by content type

Placeful workflow will help us if we need some special workflow configuration for a specific branch of folders of our intranet. However, it must be used with care and one needs to inform the users about the change of behavior of the workflow whenever it occurs.

Best practices

Understanding the complexity of the Plone workflow engine is a big deal. Here are some ideas to leverage the learning process. Otherwise, we will find that the best way is to play around and have some fun workflowing.

Make an initial blueprint first

Make a state diagram that defines our desired workflow. Not a very detailed one, but enough to accommodate states, transitions, and actions that they may trigger. Take as reference the ones included in this chapter.

Avoid developing on production servers

A classic; do not develop anything on production servers. Do it in a controlled development environment.

Start from an existing workflow copy

One of the best pieces of advice is not to begin a new workflow from scratch. If we want to create our own workflow, it is recommended to begin from a copy of anyone of Plone's out-of-the-box workflows. Start from a workflow that matches most of our desired specifications. Then go to the **Contents** tab in the `portal_workflow` view and use the controls to copy and paste the selected workflow. Rename it to a convenient name and try to describe what it will do in the description. It's always good practice to document all the things that we customize in the best way possible.

This will give us an exact copy of the original workflow and now we can begin to customize it as desired.

Use the tools shown for debugging

Use generated graphs or export the result of our workflow to CSV to debug problems and check the consistency of the whole permission map. Use the sanity check of `collective.wtf` to make sure the workflow is usable and that it doesn't have any significant flaws.

Test our workflow

Try to develop a battery of use cases in which our workflow may be used and test it against our intranet. It's easy to make a mistake related to roles and permissions, so be careful before deploying the workflow to production.

Summary

This chapter has been exclusively dedicated to workflows, best practices, and how to create, manage, and modify them.

Now, we should know:

- All of the things necessary to manage workflows in Plone
- How to create and modify workflows
- How to use some key applications related to workflows

In the next chapter, we will learn how to put all Plone's security-related subjects into practice.

7

Securing our Intranet

Now we have all the required knowledge to build a good security policy on our Plone intranet. We've already learnt about the following key concepts:

- Users and groups
- Global and local roles
- Permissions
- Workflows

This chapter will show us how to glue them together and make them work for our intranet. A good combination of all of these will allow or deny the access of content for our intranet users. Learning how to apply security effectively is essential to have full control over our intranet. Security is a shared responsibility among intranet administrators, information creators and consumers, and the content privacy managers. Any actor of our intranet has to be involved with it and should have a good knowledge on how to manage security in the intranet.

We will learn about the following topics:

- How to design a security policy
- The use of global roles and local roles
- How to manage the private content
- The private sections
- How to set up project areas for workgroups

This chapter will cover all the key concepts to build a good security policy for our intranet. This policy should be robust, usable, and reliable, and the most important: it should meet our requirements. However, the combinations and the use cases are infinite. Take the examples and best practices described here as a starting point.

Remember, in *Chapter 5, Managing Users, Groups, Roles, and Permissions*, I asked you to envisage your future intranet, especially, the security requirements it should have. Keep these thoughts in mind during the entire chapter.

Global or local roles?

The first challenge we will have to face is the types of roles we will give to our users. Let's review the properties of both of them.

Global roles are valid throughout the intranet and cannot be overridden by any means. We can manage them using the **Users and Groups** control panel configlet.

Local roles are valid only in the context they are defined in. This context could be any content type, but it's usually used to apply the type to folders, as the children objects inherit all the features of the local roles defined. We can disable the inheritance by deselecting the checkbox **Inherit permissions from higher levels** located in the **Sharing** tab. Otherwise, we can't assign some roles from the **Sharing** tab, by default. As we've already learnt, we can only assign the following roles:

- Can read (Reader role)
- Can edit (Editor role)
- Can add (Contributor role)
- Can review (Reviewer role)

The rest of the roles (**Member, Manager, Anonymous, Owner, Authenticated**) can only be assigned globally, by default. Otherwise, to make them useful, we could modify the Plone source to make them visible in the **Sharing** tab; however, it's not recommended. We will cover that at the end of this chapter.



Anonymous, Owner, and Authenticated are functional roles. Although they can be assigned to users or groups, this will make no sense.

The exclusive use of one type of role is not advisable even on the more simple, low security profile intranet. It would only be suitable in a less crowded intranet where there is no place for confidential or private sections, and all content is accessible by all users.

The answer to the question, "global or local roles?" is that we can use both of them. Using the advantages of both will be the beginning of a robust security policy.

Using global roles

Use global roles to assign structural roles that would be valid for the entire site. For example, assign the Manager role globally to only those users who will be site administrators or site developers, and assign the Editor role to those users who may need its functionality unconditionally throughout the site. Although Plone does this for us, we need all intranet users to have the Member role, thus this role must be assigned to all the users globally. Some important settings and features assume all intranet users should have this role, for example the access to the user's dashboard.



The user registration form will automatically assign the Member role to all new users, but we should assign it manually if we use users or groups from an external repository, such as LDAP or Active Directory. We can usually configure the PAS plugin for these kinds of repositories to assign the role automatically. More information on how to configure a LDAP-based user database is available in *Chapter 13, Deploying our Intranet*.

Using local roles

Use local roles to grant access and edit permissions to site contents. The roles are additive, so play with all possible combinations to extend a user's or group of users, permission to achieve the required security policy for them.

For example, if a user is assigned the Contributor role and the Reader role on a folder, the user will be able to add content and have full rights over the content he may create. This is because by being the creator of the content, the user acquires the Owner role automatically, and, in consequence, gains full access rights on the content. Otherwise, the user will have no rights on the content that is not created by him. The user will only be able to read other content if he is granted the Reader role on the parent folder.

Designing a sustainable role policy

In this section, we will propose a concrete security policy. This does not mean that this policy is the best approach to an intranet. We will have to build our own security policy based on our own intranet requirements. Consider the following example.

A policy example

Now we have the tools to design a good role policy. The policy will account for the following requirements:

- As Managers of the site, we need to have unlimited access to the site in any case and situation.
- As system administrators of the Zope instance, we must have unlimited managerial access.
- As a user, the maximum level of permission will be determined by the rights granted, by having the four possible roles that can be locally assigned and shared: Contributor, Editor, Reader, and Reviewer. Thus, a user assigned with the four roles in a particular context will reach the maximum rights in that context.
- As an Editor, we must have access rights to some special permission that grants access to some Plone features only reserved for Manager users. For example, access to manage portlets.
- As an Editor, we can add more editors to the local roles of that context.
- As an Editor and a Contributor, we can add more contributors to that context.
- As an Owner of a particular context, we must be able to see the roles assigned to that context and be able to add or modify them.
- As an anonymous user, we will have no access to the intranet. Thus, all possible access will be authenticated.

Restricting the use of the Manager role

The introduction of a new role schema since Plone 3 opened a unified (and sustainable) way to assign roles to our users. Before Plone 3, the quickest way to add edit permissions to a user was by assigning him the Manager role in the desired context. This policy is not recommendable and in most times is overkill. The Manager role has a lot of power, even if assigned as a local role. We can't assign the Manager role using Plone UI by default, but we can do so via the **Security** tab in the ZMI view of any folder. It would allow access to ZMI (restricted, if it is assigned in a particular context). But if we assign it as a local role in the Plone site root, then the user becomes manager of the site with full access to ZMI and Plone control panel.



Use this rule of thumb

Give the users only the rights they need. Of course, we can still use the *all-editors-are-managers* policy and give all editor users the manager role if we find it useful and if it fulfils our requirements, but this is definitely not advisable as it would allow the user to override parts of Plone with its own malicious code.

We can play with the permissions granted to existing roles for our security policy. Otherwise, if we still need to grant users some permission restricted to the Manager role, then we can create an additional role and grant these desired permissions (for example, Power User role).



We have covered how to create a new role in *Chapter 5, Managing Users, Groups, Roles, and Permissions* in the *Administering roles via ZMI* section.

Other approach is to grant these additional permissions to some other existing role. We can grant them the Editor role which has more power only after the Manager role. However, only do this if this role still represents editors. Create as many roles as we need to describe functionality, but manage their combinations using groups.

Creating system administrator users for the Zope instance

Let's implement the first requirement of the proposed security policy. By default, we have the *admin* user created when we set up our instance for the first time. This user lives in the `acl_users` tool of the ZMI root and it's already assigned to the Manager role. If we need more than one Zope instance system administrator, then we will have to create them in the `acl_users` tool of the ZMI root and assign them to the Manager role. These users will have unlimited access to the Zope instance, and among other things, will have access to all Plone instances inside the Zope instance.

Creating additional manager users of the Plone site

Create users, if needed, and assign them the Manager role in the **Users and Groups** control panel configlet. The manager users will have unlimited access inside the Plone site only, and would not be able to manage the Zope instance or other Plone sites inside the same Zope instance.



Separating the two concepts of system administrators and manager users of a Plone site is usually useful as we can have these two kinds of roles in our organization.

Granting other role permissions restricted to Managers

The idea of the next requirement is give the Editor role, or other roles, some additional permissions normally owned by the Manager role. Let's add permissions to allow editors to manage portlets. Go to the ZMI Plone site root and click on **Security** and search for the permission **Portlets: Manage portlets**:

Acquire?	Anonymous	Authenticated	Contributor	Editor	Manager	Member	Owner
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Select the checkbox of the **Editor** role. We can do the same for other permissions to extend the rights of the users who have the Editor role in our security policy.



We can do this programmatically and in a more reusable fashion using Generic Setup tool, by setting up a `rolemap.xml` profile. More information on this is available in *Chapter 10, Basic Product Development*.

Local role delegation

Users can manage local roles by delegation. By default, a user who is assigned the Editor role for a context can access the sharing form and assign more editors and readers. This user will be able to assign the Contributor role if the user has this role in that context. The same applies to Reviewer role. By default, content owners can define the local role assignments for all the content they have created in the site.

Allowing non-managers to administer local roles

If required, we can delegate the management of local roles granularly (one by one or all of them at the same time). This is the case of **Sharing page: Delegate roles** permission. We can delegate this behaviour to other roles (for example, the Editor role) using the ZMI **Security** tab in the desired context. We only have to enable any of the following permissions to the desired role.

- Sharing page: Delegate roles
- Sharing page: Delegate Contributor role
- Sharing page: Delegate Reader role
- Sharing page: Delegate Reviewer role



Be very careful with this setting and watch out for all implications and use cases it may have.

Choosing a workflow for our intranet

Once our role policy is in place, we should choose a suitable workflow for our intranet. Maybe one of the out-of-the-box workflows supplied with Plone may fit our needs, but mostly the probability is that they don't. It is recommended that we build our own workflow following the requirements of our intranet. There is a probability that we may need more than one workflow for different parts of our intranet. Follow the best practices shown in *Chapter 6, Managing Workflows* to build them.

Restricting access to authenticated users

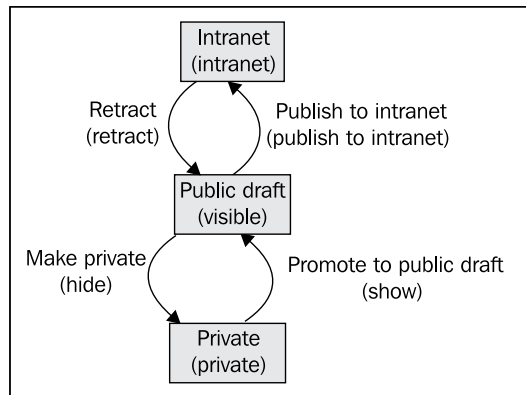
No matter what workflow we choose for our intranet, we must make sure that access to anonymous users is not allowed. We can do that by modifying the workflow and restricting the Anonymous role from any permission definition from each state in the workflow.

For example: let's assume that we choose the simple publication workflow for our intranet. Access the **portal_workflow** tool in the ZMI, and then access the `simple_publication_workflow` definition in the **Contents** tab. Then we access the published state and click on the **Permissions** tab. We must deselect all the checkboxes for the Anonymous role. This workflow doesn't allow the content to be accessed anonymously by any other state, but if we choose another workflow, we must make sure that no state has enabled anonymous access for any permission.

Then, a last step is required. We must apply the changes to the existing content in our site by clicking on the button **Update security settings**, located at the bottom of the **portal_workflow** tool. Otherwise, permissions over existing content will not be updated accordingly to new workflow definition.

Building an example intranet workflow

The first thing to do is draw the state diagram of the desired workflow. It has to be simple but it should include states and its transitions. Following is the state diagram of our example workflow:



As we can see in the diagram, we are designing a very simple workflow with three states: Private, Public draft, and Intranet. Following are the requirements for each state:

- Private content can only be accessed by the owner of the content. Only owners can change the state of an object to the private state.
- Public draft is the initial state of this workflow. All content assigned to it will be created initially in this state. Draft content can be accessed by owners, editors, and managers in the context of the object. Other intranet users can't access content in this state.
- Intranet is the state where any intranet user can see it, whereas the anonymous users can't access it.

Since this is our first workflow, let's start by copying one of the existing workflows and make the modifications needed to reach our specifications.

Start by copying the community workflow for folders, since it's the closer out-of-the-box workflow that meets the requirements. Go to **portal_workflow | Contents** tab and select **folder_workflow (Community Workflow for Folders)** and then select **Copy**, and then select **Paste**. A new workflow will appear with the name **copy_of_folder_workflow (Community Workflow for Folders)**.

Rename it by selecting it and clicking on **Rename**. Let's call it `myintranet_workflow`. Then click on it to start modifying its properties. Redefine the **Title** and **Description** as required. For example, use `My example intranet workflow` for the **Title** and set **Description** to `A workflow having three states: private, draft and intranet`, and then save the changes.

We will redefine states as shown in the state diagram and also redefine the permissions defined by them. Then, if needed, we will proceed to adjust its transitions.

Private state

Go to the **States** tab and start redefining its original states. Let's start with private state. We will keep reusing its basic properties, but we will have to redefine its permissions. Click on the private state and go to the **Permissions** tab. Modify them so that only owners and managers can access and modify content in this state. Remove the other roles assigned to the permissions:

Workflow State at /sharing/portal_workflow/myintranet_workflow/states/private

When objects are in this state they will take on the role to permission mappings defined below. Only the [permissions managed by this workflow](#) are shown.

Permission	Roles
Acquire permission settings?	Anonymous Authenticated Contributor Editor Manager Member Owner Reader Reviewer
<input type="checkbox"/> Access contents information	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/> List folder contents	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/> Modify portal content	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/> View	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>

Save Changes

We must save the changes once we have finished. Remember that we can always add new permissions managed by this workflow using the **Permissions** tab of the workflow or by following the link **permissions managed by this workflow** on this page. However, it is not necessary to fulfill the requirements of this workflow example.

Draft state

The original ID for this state in the original workflow is **visible**. We can either change it or leave it, as its title (used as the state display name in Plone UI is `draft`). We will reuse its default properties, but we should modify its permissions. Click on it and go to the **Permissions** tab. Modify them to let editors, owners, and managers view and modify content in this state. Remove the other existing roles:

Properties		Permissions		Groups		Variables	
Workflow State at /sharing/portal_workflow/myintranet_workflow/states/visible							
When objects are in this state they will take on the role to permission mappings defined below. Only the permissions managed by this workflow are shown.							
Permission	Roles						
Acquire permission settings?	Anonymous	Authenticated Contributor	Editor	Manager	Member	Owner	Reader Reviewer
<input type="checkbox"/> Access contents information	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="checkbox"/> List folder contents	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="checkbox"/> Modify portal content	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="checkbox"/> View	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input type="button" value="Save Changes"/>							

We must save the changes once we have finished.

Intranet state

We cannot reuse the published state in our example workflow but we can use it as a template. We must also disable it and delete it when we finish. We should create a new state called "intranet". This state allows all intranet users to access content assigned to it. Use the form at the bottom of the **States** tab to create a new state. Once created, access it and change its properties, such as **Title** and **Description**. Then we should change its permissions as well. Click on it to access its properties. Add a title to it because it will be used as the state display name in Plone UI. Add a description too, if you like.

Access the **Permissions** tab and uncheck the **Acquire permission settings?** checkboxes for all the managed permissions. Then allow readers to access the content and allow editors, managers, and owners to modify them too.

Workflow State at /sharing/portal_workflow/myintranet_workflow/states/intranet

When objects are in this state they will take on the role to permission mappings defined below. Only the [permissions managed by this workflow](#) are shown.

Permission	Roles
Acquire permission settings?	Anonymous Authenticated Contributor Editor Manager Member Owner Reader Reviewer
<input type="checkbox"/> Access contents information	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/> List folder contents	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/> Modify portal content	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/>
<input type="checkbox"/> View	<input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>

Save Changes

We must save the changes once we have finished.

Transitions

We should add some transitions to our recently created state. Go to the **Transitions** tab and add a new transition called `publish2intranet`. Click on the new transition and modify its properties, as shown in the following screenshot:

- **Title:** Publish to intranet
- **Description:** Publish this content to make it available to all intranet users.
- **Destination state:** intranet
- **Guard**
 - **Permission:** Modify portal content

- **Display in actions box**
 - **Name:** Published to intranet
 - **URL:** %(content_url)s/content_status_modify?workflow_action=publishtointranet
 - **Category:** workflow

The screenshot shows the configuration for a workflow transition. The transition is named 'publishtointranet' and is set to 'Initiated by user action'. The destination state is 'intranet'. The guard is set to 'Modify portal content'. The display in actions box is configured with the name 'Publish to intranet', the URL '%(content_url)s/content_status_modify?workflow_action=publishtointranet', and the category 'workflow'. There is a 'Save changes' button at the bottom left.

Assign the proper transitions to the **intranet** state by accessing the state properties and select the **retract** transition. This will allow users to change the state from intranet to draft. Save changes to apply it.

Access **visible** (draft) state and remove the **publish** transition, as it is no longer required, and add **publishtointranet** to allow users to change the state of content from draft to intranet. Leave the **hide** transition to allow users to change the state of content from draft to private.

Access **private** state and remove the **publish** transition too. Leave the **show** transition to allow users to change the state from `private` to `draft`.

Delete the **published** state if you find it's no longer useful. We are not going to use it in this example any more.



This workflow definition is included in the support code for this chapter. It's defined with a Generic Setup profile. Install the add-on product included to apply the profile to our site. After this, the workflow should be available to be used in our site.



Defining permissions, roles, and workflows with Generic Setup

Plone provides a way to define permissions, roles, and workflows, and makes them available to be reused when setting up Plone sites. It's done via two Generic Setup profiles: `rolemap.xml` and `workflows.xml`. We will cover the features of Generic Setup and other useful profiles in *Chapter 10, Basic Product Development*.

Managing private content

There are two ways of managing private content in our intranet. Both of them are valid, we may choose the one that fits better to our requirements.

The first one is, of course, workflow related. We've already noticed that most of Plone's out-of-the-box workflows come with a private state. But user's access to content in this state is not always the same. For example, in the simple publication workflow either contributors, editors, readers, owners, and managers can view and access the content in this state.

If we want to control the access to private content through the private workflow state, we should adjust permissions for making sure only the desired roles can access the content in this state.

The more common setting is to allow only the Owner role to access the private state. But our requirements may also allow other kinds of roles to access the private role. For example, we might also want to allow the Editor role to access private content.



Attention!

One must be careful. Don't forget to add the Manager role to the permission for private content. Otherwise, as an intranet manager, we will lose all privileges over the content in the private state. (However, we can regain access to the content by modifying the workflow and updating workflow settings.)

The other alternative is to disable local roles inheritance in the content we want to make private. Although it is not the best alternative, we can do it this way too. Use the checkbox **Inherit permissions from higher levels** to reset local permissions on the desired content and redefine permissions for it.



Disabling local roles inheritance will only be effective if roles are not applied globally in the site. A user assigned the Editor role globally will have this role unconditionally throughout the site. Disabling inheritance over content will have no effect over this user, who will continue having the Editor role despite it.

Creating private sections

We can create private sections or an entire branch of folders and develop the content of our site, similar to the way we create private content. But we must have one thing in mind; changing the state of a folder to a private state will not mean that all its contents will also be private. We should change its entire children object to the private state too. Otherwise, users will not be able to access the parent folder (because its permissions do not allow it), but they will be able to access the content contained by that folder. So any user could access the content inside the private folder if he/she knew the URL of that content. The content will be private as long as the URL will remain unknown, which is not very trustful. Maintaining all contents of a complete private branch in private is not a sustainable policy unless the private state will be our workflow initial state.

In this case, the disable local roles alternative is more powerful, because we will only set up the security of the folder once.



Only the owner of a folder (or a content) can disable local role inheritance. Otherwise, we will receive an unauthorized error, or lose all rights over that content.

Workgroup areas

If we need more control and we want to give more power to our users over our intranet private sections, we should use a local workflow policy (CMFPlacefulWorkflow). Analyze the workgroup requirements and apply a suitable workflow that meets the workgroup requirements. If there's an existing workflow, modify it or create a brand new one. Try to use different transition and state names than the workflow used in other parts of our intranet, so that our users don't get confused, and mark these special workgroup areas properly.



We covered Placeful workflow in *Chapter 6, Managing Workflows*. We will find the other aspects of its features and behavior in the chapter.

For work areas with more simple requirements, we can set up a workgroup folder by resetting the permissions and applying the required permissions on it. For example, we can create a private area for our company management team, where only the members of this team will be able to access its contents. Resetting the permissions by disabling local roles inheritance and giving access and edit permissions to the management group on that folder will hide it from all users except the management members.


Third-party add-on products

Sometimes third party add-ons are not "Plone 3-role-schema ready". This means that the default permissions set by these products do not take care of the new roles and give the rights to create and manage content to Managers. This can happen upon the creation of new content, and the product can add it in the workflow (or workflows) as well. The products that add new content types are more affected by them.

When we install a product, we must make sure it sets the correct permissions to the roles we use. We should check the add permission the new content type may use and modify it if necessary. We should also check whether the workflows the product may install are correct and meet our security requirements. Modify them if necessary.


Adding roles to the Plone UI

By default, the Plone UI exposes to the users only four roles. These roles are meant to be applied locally thorough the sharing form. In case we find these roles insufficient, we can add other roles, such as Manager or Owner (or any other custom role, which we might create). If we want to add a new custom role, we will have to create it first.

 Consider this a hack and don't use it even if your requirements push you to do so. Almost any security policy can be implemented without using these roles as local roles.

Using a custom product

We can achieve this programmatically by using a custom add-on product. Let's assume that we have previously created a custom add-on product named `my.firstproduct`.

 We cover basic add-on product development and how to create a new product extensively in *Chapter 10, Basic Product Development*.

We must add a module named `sharing.py` to the product with the following contents:


```
from zope.interface import implements
from plone.app.workflow.interfaces import ISharingPageRole
from Products.CMFPlone import PloneMessageFactory as _
class ManagerRole(object):
    implements(ISharingPageRole)
    title = _(u"title_can_manage", default=u"Can manage")
    required_permission = 'Manage portal content'
```

The `title` is the name to be displayed under the **Sharing** tab. The user must have `required_permission` to be allowed to manage a particular role.

Add this line to the `configure.zcml` file for declaring the new component:

```
<utility name="Manager" factory=".sharing.ManagerRole"/>
```

We will have to add this product to our buildout to apply changes.

 We can find this custom add-on product in the support code for this chapter.

Using `collective.sharingroles`

This is an add-on product that allows us to define a Generic Setup profile in a custom add-on product that handles the roles shown in the **Sharing** tab. To install it, we must add it to the `buildout.cfg` file in the `eggs` attribute of the `buildout` section:

```
[buildout]
...
eggs = collective.sharingroles
```

Also, add it to the `zcml` attribute of the `instance` section:

```
[instance]
...
zcml = collective.sharingroles
```

Then, run `buildout`:

```
$ ./bin/buildout
$ ./bin/instance fg
```

The Generic Setup profile file name should be `sharing.xml` and must be placed in the `profiles/default` folder of our custom add-on product. It should have the following format:

```
<sharing xmlns:i18n="http://xml.zope.org/namespaces/i18n"
  i18n:domain="plone">
  <role
    id="CanDelegateRoles"
    title="Can delegate roles"
    permission="Manage portal"
    i18n:attributes="title"/>
</sharing>
```

The `id` must match an existing role and the `title` is the name to be shown on the **Sharing** tab. The `permission` is optional. If assigned, the user must have the permission to be allowed to manage the particular role.



We should install our custom add-on product for the GS profile to be applied. We can also find this example in the support code for this chapter.

Summary

We have covered a best practice section, explaining things based on experience with users and basic intranet needs. We have learnt the following topics:

- How to design a security policy
- The use of global roles and local roles
- How to manage the private content
- The private sections
- How to set up project areas for workgroups

At the end of this section, we will be able to consistently apply all the knowledge about security that we have covered in previous chapters. Next, we will cover how to use Plone's content types and views effectively in an intranet.

8

Using Content Type Effectively

Building a successful intranet is not an easy job. When we are asked to build an intranet there is always an implicit requirement that doesn't show in any requirement list. This requirement is easy in concept, but hard to achieve: the intranet must be a success in terms of usability and use. We will want our users to love using our intranet and have a positive valuation of the service it offers. The last thing we will want is that it ends up being one of those web services that people barely use.

Building a successful intranet is not an easy job, but we can make it so if we want to. In this chapter, we will cover the key factors for the success of an intranet and the effective use of the content types.

As we already know, a content type is not only an information container; it also defines the way the information is shown to the consumer via the content view. We will learn to use the right content type and its right view for the right job.

Another crucial factor is to extend wisely our default content type set via third-party add-on products. We will learn how to choose and use them correctly.

This chapter will cover the following topics:

- Navigation and taxonomy
- Collections
- Table of contents
- Next/previous folder
- Presentation mode
- Best practices for third-party content types

Designing our intranet information architecture

No one uses a knowledge system (such as our intranet) if the information stored in it is hard to find or consume. We will have to specially emphasize on thinking about not only a good navigation schema, but also a successful one for our intranet. The definition of success is different for every interested group, organization, enterprise, or any kind of entity our intranet will serve. There are a lot of navigation schemas we may want to implement, but it is our task to find out what will be more suitable for our organization.

To achieve this, we will have to use both hierarchy and metadata taxonomy wisely. Obviously, the use of folders and collections will help achieve this endeavor. The first-level folders or sections are very important and we will have to keep an eye on them when designing our intranet. Also, we should not forget the next levels of folders, because they have a key role in a success navigation schema.

The use of metadata, and specifically categorization of content, will also play an important role in our intranet. The continuous content cataloging is crucial to achieve a good content search and the users should be made aware of it. An intranet where the search of content is inefficient and difficult is an unsuccessful intranet, and with time, the users will abandon it.

At this point, we should analyze the navigation needs of our intranet. Think about how the people will use it, how will they contribute contents to it, and how will they find things stored in it. In this analysis, it is very important to think about security. Navigation and security are closely related because most probably we define security by containers.

There are some standard schemas: by organization structure, by process, by product, and so on. By organization is the most usual case. Everybody has a very clear idea of the organizational schema of an enterprise or organization, and this factor makes it easier to implement this type of schema. In this kind of schema, the first-level sections are divided into departments, teams, or main groups of interest.

If our intranet is small and dedicated to one or few points of interest, then these must take precedence over the first level section folders.

Keep the following things in mind:

- Our intranet will be more usable if we can keep our intranet sections clean and clear
- Fight against those people who believe that his (or her) department is more important than others and want to assault our intranet sections
- Let them know that maintaining a good intranet structure will be more useful and will help contribute to its success

Second levels are also very important. They should be perdurable in time, interesting to users of all sections, and they should divide information and contents clearly. Two subsections shouldn't contain elements of the same subject or kind. For example, these might be a typical second level:

- Documentation
- Meetings
- Events
- News
- Forums, tracker, or some application specific to the current section

All of these are very commonly seen in an intranet. It is a good practice to create these second-level sections in advance, so that people can adapt to them.

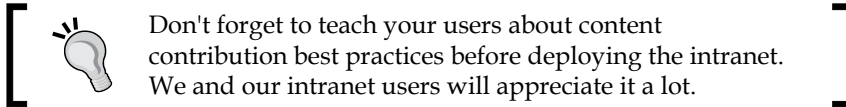
Teach people to categorize content. This will help intranet searches incredibly and will help create collections and manage contents more effectively. If needed, make a well-known set of categories publicly available for people to use. This would prevent the repetition of categories and the rational use of them.

Notice that there can be several types of categories:

- **Subject:** Terms that describe the subject of the content
- **Process:** Terms that identify the content with the organizational process
- **Flags:** Flags such as Strongly Recommended
- **Products:** Terms from the products, standards, and technology names that describe the subject matter of the resource
- **Labels:** Terms used to ensure that the resource is listed under the appropriate label
- **Keywords:** Terms used to describe the resource
- **Events:** Terms used to identify events which are recurrent with the content

There are other metadata also which influence the improvement of the navigation and search abilities of the intranet such as:

- Title
- Description
- URL, the ID of each content



Once we have settled down on some practices which are best for information architecture, we should know how to use some interesting Plone features that will help us build navigation and sort the information on our intranet.

Using collections

The collection is one of the most powerful content types available in Plone. It's probably the most misused of all Plone's default content types because it is more complex and difficult to understand for non-technical or non-experienced users.

A collection is a real-time query for the ZODB (for the Plone portal catalog, to be more precise), and its contents are the results of this query. The query is defined by the user, who can also define how to display the results in the collection view. The query is executed with the rights and context of the current user, and the results are consistent with the user's rights over content. This means that a collection could return different results depending on the current user. Additionally, a collection is a content type, and, like any content type, has permissions and is assigned to a workflow. Thus, a collection can be for both public or restricted use.

We can use them to store recurrent queries in the database, for example, the News and Events, Plone's default view, is implemented using a collection. In both cases, we have a recurrent task of displaying all published news or events. The collection will collect all news or events objects published at that precise moment and will display them. In addition, the events collection will provide an additional filter for omitting all events that occurred in the past.

I can bet you can think of other useful use cases, such as a collection, that returns all the content authored by yourself to keep track of it easily. Or maybe a collection that returns all content contributed by the members of a team or workgroup in the last month (or year). The possibilities are infinite, of course. It's in your hands to find the right use case suitable for your need.

Use collections every time you need a non-hierarchical display of content and access the information in a direct and grouped way.

Creating a collection

A collection is as easy to create as any other content type, but there are some concepts on fields and configuration that are worth elaborating. There are two places where we can configure a collection: in the edit mode and in the **Criteria** tab.

The edit mode holds all the common content type default fields along with the following:

- **Number of items:** The number of items to be shown in the results. Related to the "Limit Search results" setting.
- **Limit Search Results:** If this is selected, the results will be restricted to the number defined in the **Number of items** field
- **Table Columns:** Defines the columns to be displayed in the tabular view
- **Display as Table:** Displays the results in tabular way, with the columns defined in the **Table columns** field

But the most important setting of a collection is the **Criteria** tab. Here, we define the query on the database based on the fields or attributes of the target objects and its values. The criteria form is divided into two sections: the **Add New Search Criteria** and the **Set Sort Order**.

In the first one, we define the criteria. In order to do so, we need to inform what would be the object's field or attribute to search for and the value that we want to match the criteria with. This is usually done in two steps. First, define the field and the criteria type. For example, in **Field name** select **Categories**, and in **Criteria type**, select one of the three options that will determine how we will define the criteria: select it from a compiled list from all the available categories in the site (**Select values from list**), type a single category into a text box (**Text**), or type a list of categories separated by a carriage return (**List of values**).

Once we select one of them, the form will change to add the new criteria. The second step will consist of defining the category (or categories) that the criteria will match with. Almost all the criteria have this two step process. We can add as many criteria as we need. Don't forget to save any change made to the criteria settings.

Criteria for A collection

No criteria defined yet. The search will not show any results. Please add criteria below.

Add New Search Criteria

Field name
The keywords used to describe an item
Categories

Criteria type
Criteria does match
Select values from list

Add criteria

Set Sort Order

Field name
List Available Fields
No sort order

Reverse
Reverse display order

Save

The second part of the form is related to the ordering of the results. This order can be set against a field name and can be specified to be in the reverse order, if desired. Reverse order is useful when ordering according to dates, the most recent on the top of the collection's results.

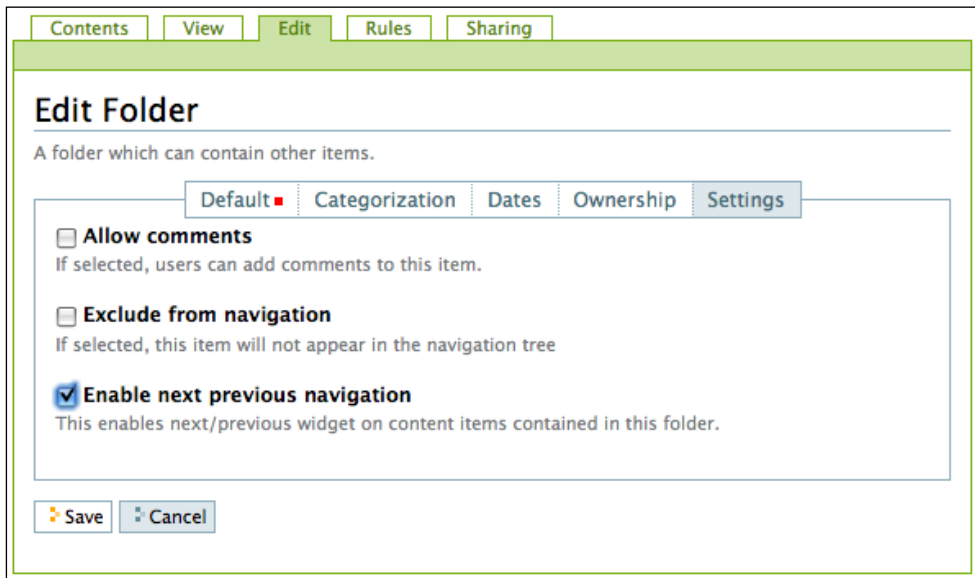
Table of contents

This is a useful feature of the page content type. We can enable it through the **Settings** tab in the **Edit** mode. It adds a handy table of contents menu to the top right of the default view of the current page content type. It's formatted using the headings of the contents of the page, and is created and updated automatically. It also features relative links to the headings of the page. It's very useful on very large pages, where we want to keep access to contents clear and quick. We can see the result in the following screenshot:

The screenshot shows a page editor interface with a green header bar containing tabs for 'View', 'Edit', and 'Sharing'. On the right side of the header, there are 'Actions' and 'State: Private' dropdown menus. The main content area displays the page title 'My first page', the author 'admin', and the last modified date 'Jan 09, 2010 07:21 PM'. Below this is a placeholder for a description and a paragraph of rich text content. A sidebar titled 'Contents' is positioned on the right, listing the page's sections and subsections with relative links. The sections listed are 'First section', 'Second section', and 'Third section'. The 'Second section' is further detailed with two subsections: 'First subsection of the second section' and 'Second subsection, second section'.

Next/previous navigation

We can enable horizontal navigation thorough all elements of a folder via this feature. It's useful in case we have a lot of information to show on a single page. We can divide this content into smaller pieces in order to make it more usable, clear, and searchable. Put each section into different pages inside a *thematic folder* on the subject we are writing (name it conveniently), and check the **Enable next previous navigation** checkbox in the folder's **Settings** tab in **Edit** mode, as shown in the following screenshot:



When we access any content in the folder, we will be able see the additional controls to navigate to the next or previous item in the folder. It works with any content type existing in the folder. The title of the next or previous item will also appear on the next/previous controls.

Presentation mode

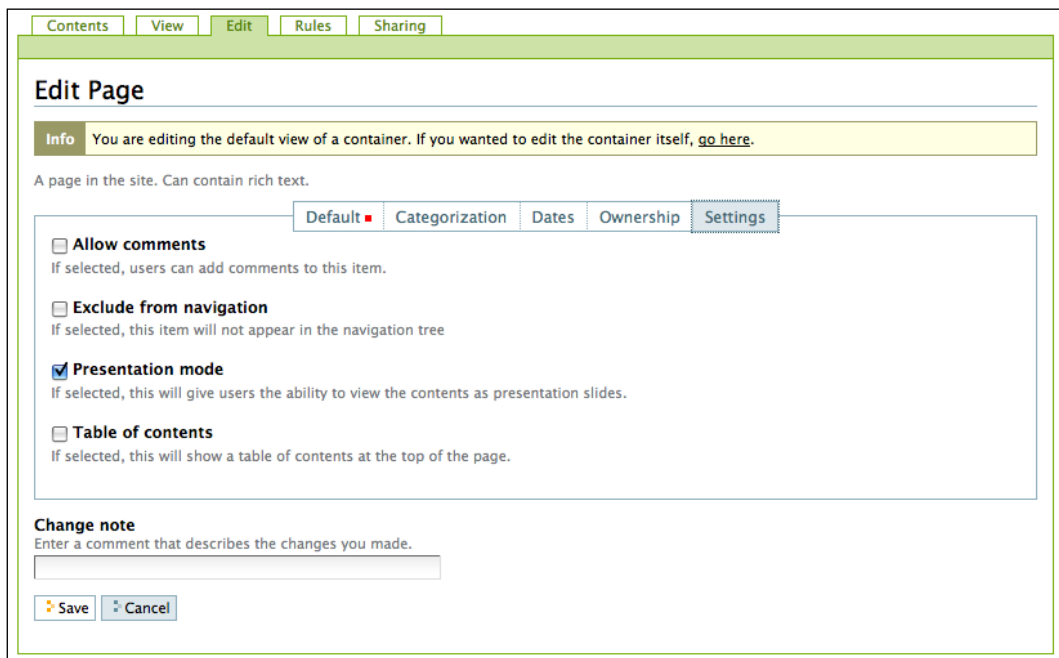
Meet one of the most unknown features of Plone 3 and one of the more appreciated features by management staff. This feature enables any page content type to make available a special view called **Presentation mode**. This view powered by the **S5 JavaScript library** shows each section of the page in a presentation-like slide. So we can easily create a content page both for documenting and for easily showing its content in a projected presentation, all in the same place as all the information lives in the same piece of content.

The sections are delimited by the heading styles included in a page, so we don't need to create a page for each slide. In fact, the heading style in Kupu or TinyMCE will be transformed to a h2 HTML tag that will be used by S5 to format the presentation. All content between h2 tags will be formatted as slides. The content of the h2 tag will be the slide's title and the content located between a h2 tag definition. The next one will be the slide's content.

A leading slide will be added automatically with the title, description, and author of the page. The S5 engine will also render basic controls over the presentation itself for navigating around the presentation's slides. Then it will close the presentation mode and return to the normal visualization mode.

Enabling the presentation mode

We can enable presentation mode by editing any page content type and then clicking on the **Settings** tab. The **Presentation mode** checkbox is available there. Once checked, a link will appear in the view of the page for a user to view the page in Presentation mode.




The screenshot shows the 'Edit Page' interface with a navigation bar at the top containing 'Contents', 'View', 'Edit', 'Rules', and 'Sharing'. Below the navigation bar is the 'Edit Page' title and an 'Info' box stating: 'You are editing the default view of a container. If you wanted to edit the container itself, [go here](#).' Below this is a description: 'A page in the site. Can contain rich text.' A horizontal tab bar contains 'Default', 'Categorization', 'Dates', 'Ownership', and 'Settings', with 'Settings' selected. The 'Settings' panel includes four options: 'Allow comments' (unchecked), 'Exclude from navigation' (unchecked), 'Presentation mode' (checked), and 'Table of contents' (unchecked). Below the settings is a 'Change note' section with a text input field and 'Save' and 'Cancel' buttons.

Formatting a slide

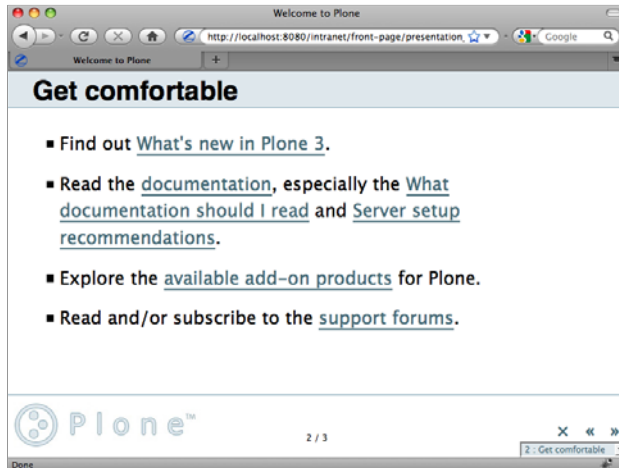
Following the most popular practices on how to construct a presentation slide, the presentation mode will not render text chunks or paragraph text. Slides are meant to display concepts, ideas, and summary information. For this reason, if we want to display content in the Presentation mode, we must format it with a style other than the *Normal paragraph* style. For example:

- Subheading
- Definition list
- Bulleted list
- Numbered list
- Literal
- Pull quote
- Highlight (if not inside a paragraph)

We can add images too, if they are not inside a paragraph tag. It's possible that we may want to hack the HTML code in order to achieve the best results if we want to format a complex presentation. We can do this by triggering the Kupu's HTML view button. Remember that all content inside a `<p>` tag will not be rendered. If you want to make some previously created content presentation mode ready, it would possibly require some minor adjustments before it will show properly.

 We can use this feature to add additional support information to the presentation in our page that will not be rendered in Presentation mode. By doing this, with smart usage of page formatting, we can write a page with two purposes: holding the detailed documentation of a particular subject, along with the presentation that consists of the summary and highlights of the subject.

We can take the default **Welcome to Plone** page as an example on how to proceed with Presentation mode. The following screenshot is the third slide of Plone's default page in Presentation mode view:



Third-party content types—best practices

We've already learnt about how to use Plone's default content types wisely. Now is the time to talk about third-party content types provided by third-party add-on products.

Sooner or later, we will find ourselves browsing the downloads section on the Plone site and will be tempted to try a lot of products that promise wonderful features and incredible content types. We recommend you to follow some rules when dealing with third-party content types—firstly, don't rush into it and be careful.

A few golden rules

We should observe some golden rules before a third-party product is put in production. They are valid for all types of third-party products and not only for those that provide a new content type, of course. They are very simple and can save us from trouble:

- Find out who's the author (or authors) of the product, and how many other contributions they have made to the community
- Check out if the product is uploaded to the SVN collective repository (<http://svn.plone.org/svn/collective>), to make sure that all the members of the community can contribute and improve it
- Check how long the life cycle of the product is and if it's a final release

- The product must have decent documentation that would lead us to a better understanding of what the product does (and what it *doesn't* do)
- Check if the product has enough test coverage
- Always test the product in a development environment, and if possible test it with real data

Ordering the "Add new" content type menu

By default, all installed content types are shown in the **Add new...** menu, ordered alphabetically, but we can restrict the types of content that can be added. This menu has two configurable levels. The first level is the menu that unfolds when we click on the **Add new...** tab and shows the allowed types. The secondary one is a view that can be accessed from the item **More...** of the first level drop-down menu:



This allows us to separate the most used or preferred content types from the least used or less popular ones. Thus, simplifying the allowed content types drop-down menu and making it more usable at the same time.

Believe it or not, this simplification usually provides a better user experience. If we make a proper selection of the more used content types (or those which we think are more appropriate for the user to use), the users will have more possibilities to find the right content type quickly. If we give them many possibilities, the chances of choosing the wrong content type is high. The user's confidence in the intranet will drop due to not finding a suitable content type for the job they require at first glance. In the end, this simplification provides a better user experience. If the user can't find the right content type from the first selection, the user will access the extended one in the secondary types view and continue with the creation of content.

The item **More...** will be displayed when we tell Plone which content types we want to show as primary types and which as secondary types. All users with the *Manager* role on a folder (except on the site root) will be able to access the item **Restrictions....** It will be located in the new content type drop-down menu. This item will lead us to the restriction policy form. In this form, we can choose between **Allow the standard types to be added** or **Specify types manually**. The former is the default option; whereas the latter will open an additional form where we can specify which will be the allowed types and which will be the secondary ones. Specify the allowed (and primary) types in the upper part of the form and the secondary (the ones that will appear in the **More...** view) types in the lower part of the form.

Restrict what types of content can be added

Type restrictions
Select the restriction policy in this location.

Allow the standard types to be added
 Specify types manually

Allowed types
Controls what types are addable in this location.

<input checked="" type="checkbox"/> Collection	<input checked="" type="checkbox"/> File	<input checked="" type="checkbox"/> Link
<input checked="" type="checkbox"/> Event	<input checked="" type="checkbox"/> Folder	<input checked="" type="checkbox"/> News Item
<input checked="" type="checkbox"/> Favorite	<input checked="" type="checkbox"/> Image	<input checked="" type="checkbox"/> Page

Secondary types
Select which types should be available in the 'More...' submenu *instead of* in the main pulldown. This is useful to indicate that these are not the preferred types in this location, but are allowed if you really need them.

<input type="checkbox"/> Collection	<input type="checkbox"/> File	<input type="checkbox"/> Link
<input type="checkbox"/> Event	<input type="checkbox"/> Folder	<input type="checkbox"/> News Item
<input type="checkbox"/> Favorite	<input type="checkbox"/> Image	<input type="checkbox"/> Page

Content type superseding

To avoid user confusion it is recommended to not maintain two (or more) content types with similar purposes. This includes content types that have similar fields, functionalities, or views. It's a good thing that all intranet users choose the same content type to perform the same kind of job.

If we still want to add the new type to our allowed content types, it's recommended to hide the creation option of the superseded type. Of course, all the content already created with the old type will be preserved, but the user will not be able to create more content using it.

We can use the **portal_types** ZMI tool to hide the old content type. To access it, click on the content type we want to hide. Inside the type form, the **Implicitly addable?** property controls the visibility of the content type item in the drop-down menu **Add new...** Once we uncheck it, the content type will not be available to add.

The screenshot shows the ZMI interface for the 'portal_types' tool. At the top, there are tabs for 'Properties', 'Aliases', 'Actions', 'Undo', 'Ownership', 'Interfaces', and 'Security'. Below the tabs, the title bar reads 'Factory-based Type Information with dynamic views at /intranet/portal_types /Document'. A help link is visible on the right. A text block explains: 'Properties allow you to assign simple values to Zope objects. To change property values, edit the values and click "Save Changes".'

Name	Value	Type
Title	<input type="text" value="Page"/>	string
Description	<input type="text" value="A page in the site. Can contain rich text."/>	text
I18n Domain	<input type="text" value="plone"/>	string
Icon	<input type="text" value="document_icon.gif"/>	string
Product meta type	<input type="text" value="ATDocument"/>	string
Product name	<input type="text" value="ATContentTypes"/>	string
Product factory	<input type="text" value="addATDocument"/>	string
Initial view name	<input type="text" value="document_view"/>	string
Implicitly addable?	<input checked="" type="checkbox"/>	boolean
Filter content types?	<input checked="" type="checkbox"/>	boolean
Allowed content types	<ul style="list-style-type: none">ATBooleanCriterionATCurrentAuthorCriterionATDateCriteriaATDateRangeCriterionATListCriterionATPathCriterionATPortalTypeCriterion	multiple selection
Allow Discussion?	<input type="checkbox"/>	boolean
Default view method	<input type="text" value="document_view"/>	string
Available view methods	<input type="text" value="document_view"/>	lines
Fall back to default view?	<input type="checkbox"/>	boolean

At the bottom of the form, there is a 'Save Changes' button.

It's also a good practice to inform the intranet users about the change, along with the new features of the content type and any significant information they might want to know.

Maintaining usability

Maintaining the intranet usability is a goal we have to continuously keep in mind. Making a lot of complex content types available in our intranet types, which are hard to create and consume, will lead to an intranet which will be hardly used. Even the more experienced techie users will end up not using it. Think about the interests of the non-technical users and you have a lot gained.

Upgrades

If we choose the right third-party products for our intranet, it will be easier for us if we want to upgrade the product itself or Plone. If a product has heavy community support, there is a probability that it will have good upgrade options between the versions as the product evolves. The same thing happens with Plone's upgrades. The product will be ported more easily if the product is well supported.

Summary

We have to be very careful about the content types we make available for our users. Failing to do so will lead to really hard upgrades, product conflicts, and user confusion.

In this chapter, we've covered the following topics:

- How to organize information on our intranet
- How to get the maximum out of Plone features, such as collections, previous/next navigation, and so on

In the next chapter, we will go through a selection of the more interesting third-party products available in the Plone community. We will also cover the ones which are specially relevant to the intranet. All of them meet the golden rules of the third party products and are ready for production.

9

Intranet Add-on Products

This chapter covers the most useful tools that are used in an intranet. We will show the most reliable, tested, and fully featured set of Plone's third-party, add-on products. Some of them focus on improving user experience, whereas others focus on extending collaboration capabilities and daily work in a corporate environment. We will cover several product categories, such as:

- Internal blogs
- Group discussions
- Form generators
- Surveys and polls
- Calendaring and events
- Other useful intranet products

We intend to compile an initial stack of products for our intranet to complete and extend the out-of-the-box functionality of Plone. Of course, it's our decision whether to install them or not. Otherwise, we can complete this stack of products with something that meets our requirements by visiting <http://plone.org/products> or the Cheeseshop site at <http://pypi.python.org>.



Some of the product features explained here may change or may be modified as part of the product evolution. If you have any question about any product shown in this page, please refer to the product's project owner displayed on the plone.org or [PyPI](http://PyPI.org) site.

Calendaring and extended events

One of the most requested features on an intranet is to provide the intranet users with a usable calendaring system. Plone's default event, and related views, and portlets are limited in some aspects. The products exposed in this section extend Plone's default capabilities on calendaring.

Plone4ArtistsCalendar

Although this product doesn't fall under the *maintained* category, it's worthy to talk about it as its features are very useful in an intranet environment. Also known as `p4a.plonecalendar`, it allows us to turn any normal Plone folder or collection into a calendar. The calendar provides a default month, event list, and event archive view.

Installation

Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = p4a.plonecalendar
...
[instance]
...
zcml = p4a.plonecalendar
```

Then, rerun `buildout`:

```
$ ./bin/buildout
$ ./bin/instance fg
```

This package has dependencies over other Python and **P4A (PHP For Applications)** modules. The `buildout` will download and make them available to the environment.

Go to the Add-on Products control panel configlet and install it.

Features

The Plone4ArtistsCalendar product does not install any content types, but it provides the following features:

- **Calendar support:** Any folder or collection can be calendar activated. Once activated, they will show a new default view displaying a calendar with an overview of the events contained by the folder or the collection. We can activate it using the drop-down menu subtypes located in the content actions bar. We can deactivate it using the same drop-down menu to return the folder to its default behavior.

- **Monthly, weekly, and daily view:** Any calendar activated folder can have several default views, including a daily, weekly, and monthly view. We can assign any of these default views to the folder.
- **Chronological event view:** The events gathered together by the activated calendar can be displayed using a chronological event listing.
- **Past events view:** Events that have already occurred are grouped into a past events listing page.
- **iCal and hCal support:** Exporting events in iCal format and importing iCal and hCal. Publishing a calendar from Apple iCal or Mozilla Sunbird to our Plone site.

This is a calendar activated folder:

The screenshot shows a web-based calendar interface for January 2010. At the top, there are navigation tabs: 'Contents', 'View', 'Edit', 'Rules', and 'Sharing'. Below these are sub-menus: 'Sub-types', 'Actions', 'Display', 'Add new...', and 'State: Published'. The main heading is 'Prova', followed by the author 'Victor Fernandez de Alba' and the last modified date 'Jan 23, 2010 01:58 PM'. There is a link to 'Import iCal'.

The calendar view is set to 'Month' and shows the following events:

Week #	Mo	Tu	We	Th	Fr	Sa	Su
53					1 New year's Day	2	3
1	4	5 Epifany	6	7	8	9	10
2	11	12	13	14	15	16	17
3	18	19	20	21	22 Calendar sprint	23 Calendar sprint	24 Calendar sprint
4	25 Roberto's birthday	26	27	28	29	30	31
5							

At the bottom, there is a 'History' section and links for 'Send this' and 'Print this'.

Take advantage of local security and collections

Don't hesitate to take advantage of other Plone features at any time. We can use local role security to set up the user's rights over events in a folder or group of folders. We can control the visibility of an item by granting permissions granularly on the hierarchy of folders.

Think about this scenario: we want to set up a calendar view for three groups of users in our intranet. We want to show the user the events depending on the user's group membership. If the user is member of group A, he will only be able to see the events related to group A in the calendar view, and so on.

We will have to set up a main folder that should contain as many folders as the number of groups we want to add. Let's assume we have three groups: group A, group B, and group C. This folder will also contain a collection that will query the database for all events inside the main folder, including all subfolders it may contain. Then, activate the calendar view on this collection and make this collection the default view of the main folder.

Then, we must set up the permissions on each event group folder. We should do it by granting permissions to the corresponding group of users in each folder.

Now users will only be able to access those events for which they have view permissions on the calendar view.



For more information, refer the project page, on the PyPI website:
<http://pypi.python.org/pypi/p4a.plonecalendar>.

vs.event

This extends the default event content type. It adds a new content type called VSEvent. It's basically an extension of the default event content type. It provides full integration with P4ACalendar.

Installation

Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = vs.event
...
[versions]
Products.DataGridField = 1.6
```

```
...  
[instance]  
...  
zcml = vs.event
```

Then, rerun buildout:

```
$ ./bin/buildout  
$ ./bin/instance fg
```

This product depends on `Products.DataGridField` add-on product, which provides a field to store the event attendants. We have to pin the version of this product to ensure that buildout downloads the Plone 3 compatible version. Go to the Add-on Products control panel configlet and install it.

Features

It provides:

- Recurring events support
- All day events support
- Attendees and attachments
- iCal and vCal export
- Supplementary events

We can find the recurring form under the **Recurrence** tab in the `VSEvent` content type edit mode. We can set the interval for which the event should repeat as follows:

- By weekdays
- By occurrence within the selected weekdays
- We can define exceptions, if any
- The frequency with which it should repeat
- The maximum count for which an event repeats
- The date until it will repeat

Following is the recurring event form:

Add VSEvent

Information about an upcoming event, which can be displayed in the calendar.

Recurrence ▾

Weekdays
Select weekdays

Monday
 Tuesday
 Wednesday
 Thursday
 Friday
 Saturday
 Sunday

Occurrence
Comma separated list of numbers. If this event is on first and third Monday of the month, enter "1,3" and select appropriate day of the week above. For the last day of the month, enter "-1", select Monday through Friday above and monthly recurrence.

Exceptions
Please enter exceptions to recurrence. One date per line in format YYYY-MM-DD

Frequency ▾
Does not repeat ▾

Interval ▾
1

Count
Maximum number of times the event repeats

Repeat until
Event repeats until this date
 : : hour

We can restrict the default event content type from being added by the users and leave only the new `VSEvent` content type as the site-wide default event type. We can do so by using the **Restrictions...** menu in the **Add new...** drop-down and removing the event content type from the available addable types. More information about hiding a content type from users is available in *Chapter 8, Using Content Type Effectively*.



For more information, refer to the project page on the plone.org website <http://plone.org/products/vs.event> and on the PyPI website <http://pypi.python.org/pypi/vs.event>.

Form generators

Eventually, we'll need the ability to create a custom form and make it available to our intranet users. It's a very common use case; a custom form is useful to ask people things, gather user data, and so on. When we talk about form generators we are referring to a tool that allows us to create complex forms with different actions involved when the user submits the form. In this category, there is only one product that excels over any other – `PloneFormGen`.

PloneFormGen

This product provides a generic Plone form generator. The forms are built using Plone content types that `PloneFormGen` provides. It's very useful to build simple web forms that save or mail form input.

Installation

Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = Products.PloneFormGen
```

Then, rerun buildout:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Go to the Add-on Products control panel configlet and install it.

Dependencies

PloneFormGen depends on the Scriptable fields package bundle that provides these additional modules:

- Products.TALESField
- Products.TemplateFields
- Products.PythonField

They are installed automatically as the `Products.PloneFormGen` egg depends on it. The buildout process will download and install them for us.

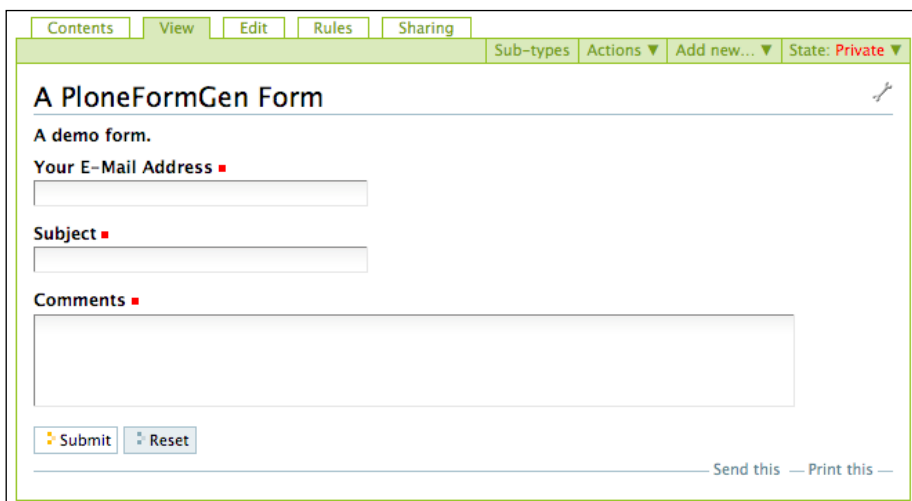
How it works

To build a web form, create a form folder, and then add form fields as contents. We can add three types of objects:

- Field types
- Actions (also named as adapters)
- Other content types

The form folder creates a form from all the contained field types content objects. The action objects inside the form folder define the behavior of the form. The form folder can also contain other types of objects that complement the form and may act as support elements, such as images or pages.

To make it easy to get started, newly created form folders are pre-populated to act as a simple e-mail response form.



The screenshot displays a web browser window showing a PloneFormGen form. The interface includes a top navigation bar with tabs for 'Contents', 'View', 'Edit', 'Rules', and 'Sharing'. Below this, there are additional controls: 'Sub-types', 'Actions' (with a dropdown arrow), 'Add new...' (with a dropdown arrow), and 'State: Private' (with a dropdown arrow). The main content area is titled 'A PloneFormGen Form' and contains a 'demo form'. The form consists of three input fields: 'Your E-Mail Address', 'Subject', and 'Comments'. At the bottom of the form, there are two buttons: 'Submit' and 'Reset'. In the bottom right corner of the form area, there are links for 'Send this' and 'Print this'.

Field types

Following is a list of all field types available in PloneFormGen:

Field type	Description
Checkbox Field	A true/false field, which will be rendered as a checkbox widget.
Date/Time Field	A field to store date/time values. Will render with the standard Plone date/time widget.
Decimal Number Field	Use this when we want a numeric response that might include a decimal point.
File Field	Allows the user to upload a file. Files will be e-mailed as attachments to a previously specified mail address by the mail adapter.
Label Field	A simple label and help text with no data-entry field.
Lines Field	A field that allows the user to submit multiple lines of text. The save-data adapter will separate lines with newline codes.
Multi-Select Field	A field that allows the user to choose one or more values from a list. Can render using a multi-select box or as a list of checkboxes.
Password Field	A field that displays the * character when the user types. It's good for submitting passwords or other sensitive data.
Rich Label Field	A field with no input, just a HTML-formatted label. Useful for adding descriptive text to our form.
Rich Text Field	A field that renders the default rich text editor widget to insert rich HTML formatted data.
Selection Field	A field that lets a user choose a single value from a list. Can render as a selection list, a drop-down menu, or as a list of radio buttons.
String Field	A field that lets a user input a single line of text. This is the standard general-purpose text field.
Text Field	A field that lets a user submit a longer chunk of text. Renders as a multi-line text box.
Whole Number Field	A field for storing a normal integer. This is the best field to use for simple numerical responses.

Action adapters

The following list shows the action adapters available:

Action name	Description
Custom Script Adapter	Allows us to write python scripts for simple form actions without having to use the Zope Management Interface.
Mailer Adapter	An adapter that e-mails form submissions to one or more users. It can optionally encrypt the e-mails using GNU Privacy Guard (GPG) encryption, if it is installed on our system. We can find more information about how to use GPG at http://www.gnupg.org/ .
Save Data Adapter	An adapter that saves form submissions inside the form. Let us view the results on screen, and download as tab- space or comma-delimited text.

Other content types in a form folder

A form folder can also contain these other content types:

Content type name	Description
Fieldset Folder	Lets us render sets of fields on our form. It's a folderish object into which we can place fields. Each Fieldset Folder will render as an HTML <code><fieldset></code> in the form. This is useful for giving our forms different sections and structure.
Image	A normal Plone image, which we may want to refer to in our form, perhaps in a Rich Label Field, a Page, or a Thanks Page.
Page	A normal Plone page. We may use these as "thanks" pages if we don't need to display input.
Thanks Page	A page, which will be displayed to the user after they submit the form. We can create multiple Thanks Pages and use logic to choose which one the user receives.

The screenshot shows the PloneFormGen interface. At the top, there are tabs for 'Contents', 'View', 'Edit', 'Rules', and 'Sharing'. Below these is a header for 'A PloneFormGen Form' by 'admin', last modified on Jan 23, 2010. A table lists form components: Mailer, Your E-Mail Address, Subject, Comments, and Thank You. A dropdown menu is open, showing various field types like Checkbox Field, Custom Script Adapter, Date/Time Field, etc.

Select: All				
	Title	Size	Modified	State
<input type="checkbox"/>	Mailer	0 kB	Jan 23, 2010 07:03 PM	
<input type="checkbox"/>	Your E-Mail Address	0 kB	Jan 23, 2010 07:03 PM	
<input type="checkbox"/>	Subject	0 kB	Jan 23, 2010 07:03 PM	
<input type="checkbox"/>	Comments	0 kB	Jan 23, 2010 07:03 PM	
<input type="checkbox"/>	Thank You	0 kB	Jan 23, 2010 07:03 PM	

Extensibility and third-party products for PFG

PloneFormGen is designed to be highly extensible. Although for an intranet use case, in the vast majority of cases the default field types will suffice, we can program new field types or action adapters very easily.

There are also third-party products that extend PloneFormGen, for example Salesforce PFG Adapter, a product to integrate Plone with Salesforce.com CRM database.

Captcha integration

It's possible to integrate a CAPTCHA (test used in computing to ensure that the response is not generated by a computer) field with the products `collective.captcha` and `collective.recaptcha`. Just make sure that these products are installed and available for Plone when we install PFG and the installation process of PFG will enable the integration with them. Install these two products as usual, for `collective.recaptcha`.

Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = collective.recaptcha
...
[instance]
...
zcml = collective.recaptcha
```

Then, rerun buildout:

```
$ ./bin/buildout
$ ./bin/instance fg
```

`collective.recaptcha` uses the Carnegie Mellon University's `recaptcha.net` service. We will need to register an account on <http://recaptcha.net> and get a public/private keypair. Registering an account for recaptcha is free. Set the keypair in the PFG configlet in the site control panel.



For more information, refer to the project page, on plone.org:
<http://plone.org/products/ploneformgen> and
<http://plone.org/products/ploneformgen/documentation>.

Blogs

There are a lot of good products that enable adding blogs in our Plone site. Although one may think that a blog is not a common application inside an intranet, it is a very usual use case. A blog is an inestimable sharing and collaboration tool. It is very valuable for communication between teams or departments inside a corporate intranet.

For example, nowadays it's common to see a corporate blog display the news announcements or product updates of the company (or a team or group). The research information generated by a specific team or the IT team updates of the company systems are also eligible to be a blog.

Here we are going to cover two options; one simple and one fully featured.

Quills

Quills is basically the blog Plone product with more history. It is designed to provide specialized features for a multi-blog, multi-user environment. It's probably the more complete blog product, but also the more complex one to use.

Installation

Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = Products.QuillsEnabled
```

Then, rerun buildout:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Go to the Add-on Products control panel configlet and install it.

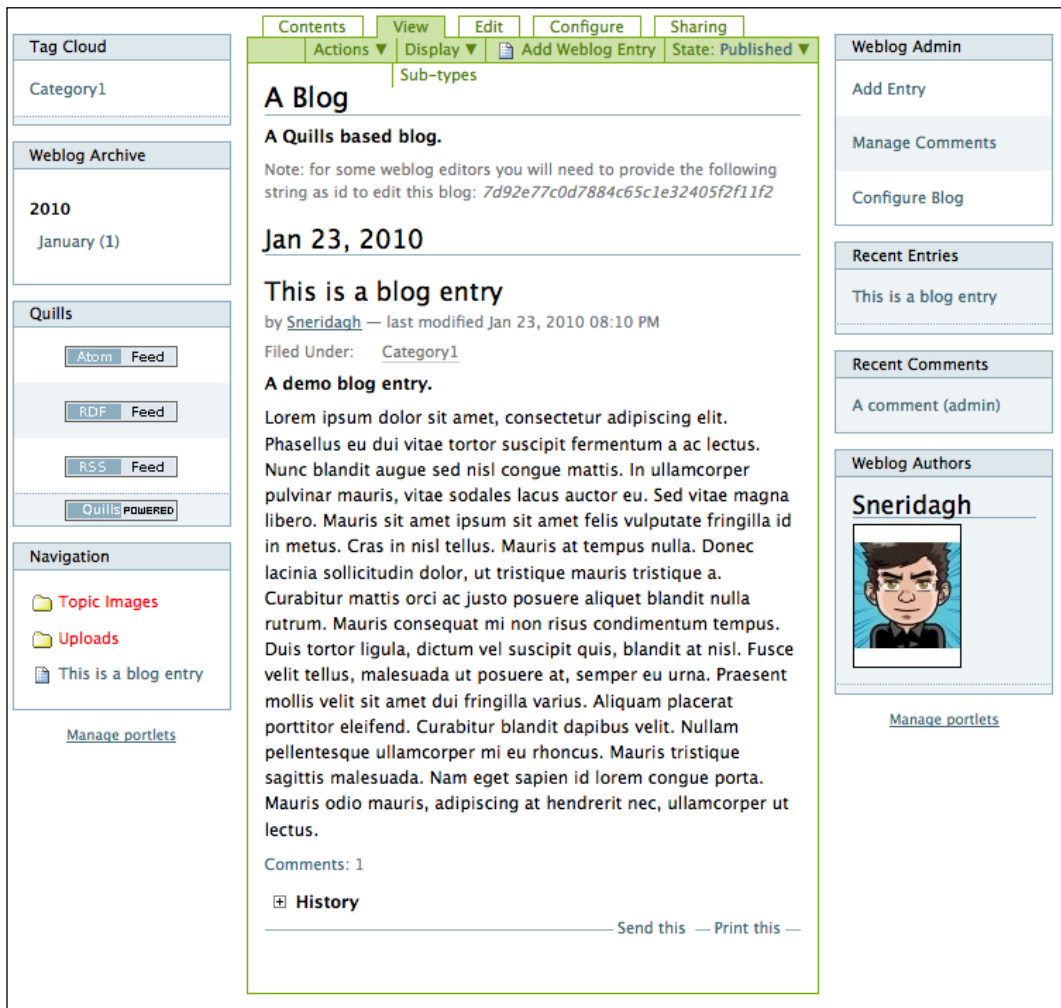
Features

To create a blog, we can simply add a new folder and select **Activate Blog** from the **Actions** menu. Existing folders can be turned into a blog in the same way.

Quills have the following main features:

- Uses Plone folders, documents, and news items as blog entries
- Uses Plone's machinery for comments
- Archival entry paths, access blog archives via standard paths, such as <http://example.com/myblog/archive/2004/04/06/>
- Topics with descriptions and images
- BloggerAPI support for remote posting
- Trackback pings
- Site-wide blog aggregator
- Blog planets for groups
- Multiple topics and advanced topic searching with simple URLs, such as <http://example.com/myblog/topics/work/project/>
- Include all the post body in the RSS feed
- Custom Quills portlets

The following screenshot shows the main view of a Quills blog:



Quills portlets

The product provides some bloggish-like portlets. As any portlet, they can be reordered, removed, and assigned to any portlet column. We can manage them through the **Manage portlets** link. Following are the Quills portlets:

- **Tag Cloud:** A list formed from the keywords assigned to each post. It has the typical visually weighted design according to the number of appearances of the keyword assigned to content.
- **Weblog Archive:** A chronological list of all the posts

- Weblog Admin: A collection of links leading to special forms and actions, such as the **Configure Blog** form.
- Recent Entries: The list of the most recent entries.
- Recent Comments: The list of the most recent comments.
- Weblog Authors: A list of the blog contributors.

Configuring the blog

Quills have a special form to set up some aspects and behavior of the product:

- Only excerpt in weblog view: When enabled, shows only the title and excerpt in the main weblog view. If an entry has no excerpt, only its title will be displayed.
- Group by dates: When enabled, entries will be grouped under a header showing the date. Otherwise, the entries will be just shown under each other.
- Entries per page: Selects the number of weblog entries we would like to display on the front page and any other batched pages.
- Show topic images in weblog view: This controls the display of topic images in the weblog view.
- Enable the receiving of traceback pings: This controls whether traceback is enabled in the weblog.
- Archive URL prefix: Allows (optionally) injecting a segment into archive URLs after the weblog segment.
- Show 'About' info: If selected, the item creator and modification date will be shown.
- Published workflow states: Workflow states will be treated as published.
- Draft workflow states: Workflow states will be treated as draft.



We can find more information in the PyPI website:
<http://pypi.python.org/pypi/Products.QuillsEnabled>.

Scrawl

Scrawl is a lightweight bloggish product for Plone. Compared to other Plone blog products, it is deliberately very minimalistic.

Installation

Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = Products.Scrawl
```

Then, remount buildout:


```
$ ./bin/buildout
$ ./bin/instance fg
```

Go to the Add-on Products control panel configlet and install it.

Features

It uses a cloned News Item content type to be used as a blog entry. It also uses a slightly tweaked view template and adds an alternative view to the collection content type called `blog_view`, available in the **Display** menu. This view either shows the description of each contained blog entry (if it exists) or the entire body in it. So it's up to the user to limit those results in collection settings, otherwise the page may take too long to load.

With these elements, plus Plone's built-in features, it is easy to quickly construct a simple, powerful blog.



We can find more information about Scrawl on plone.org:
<http://plone.org/products/scrawl> or the PyPI website:
<http://pypi.python.org/pypi/Products.Scrawl>. More
information on how to set up a blog with Scrawl is available at:
[http://plone.org/products/scrawl/documentation/
how-to/creating-a-blog-with-scrawl](http://plone.org/products/scrawl/documentation/how-to/creating-a-blog-with-scrawl).

Discussion board

An old-fashioned discussion board is always appreciated. It's true that it may be replaced by other forms of web applications used for communication, such as blogs, but in some cases, a discussion board is irreplaceable.

We are going to cover the most popular board in Plone's community – PloneBoard.

PloneBoard

PloneBoard takes advantage of Plone's machinery to work. There are other alternatives that use external relational databases, but PloneBoard stores its information in content types. It doesn't pretend to be a fully-featured message board, but will stay minimal in the sense that it's living inside an existing content management system.

Installation

Nowadays this product is under development, being revamped and adapted to Plone 4. In order to make it work for Plone 3, we should pin the version of this product to 2.0.1. Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = Products.Ploneboard
...
[versions]
Products.Ploneboard = 2.0.1
Products.SimpleAttachment = 3.3
```

Then, rerun buildout:

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

```
$ ./bin/instance fg
```

Go to the Add-on Products control panel configlet and install it. It has a dependency on `Products.SimpleAttachment`, but the buildout process will download and install it for us.

The screenshot shows a web interface for a PloneBoard forum. At the top, there are navigation tabs: Contents, View, Edit, RSS Feed, Moderate, and Sharing. Below these is a sub-header with 'Sub-types', 'Actions', 'Display', 'Add Forum', and 'State: Open'. The main heading is 'A Ploneboard forum' by Sneridagh, with a timestamp 'last modified Jan 24, 2010 01:13 PM'. There is a search bar labeled 'Search Board' and two links: 'Show all recent activity' and 'Show all unanswered'. Below this is a section titled 'General forums' containing a table with the following data:

Forum name	Conversations	Most recent comment
A new forum	1	by Sneridagh Sunday 22:21

At the bottom right, it says 'Powered by Ploneboard' and provides links for 'Send this' and 'Print this'.

How it works

Ploneboard has three content types:

- Message board
- Forum
- Conversation

The message board content type is a folderish type that contains forums. Forums in turn contain conversations which can be replied to.

Message board is the type that contains all the board information. We can set up the name of the board and the categories that the contents of the board may be assigned. From the main board view, we can configure the RSS feeds associated to the board and the moderation view, which shows all the conversations pending to be approved. It also has two kinds of default views – **Global forum listing** view and **Local forum listing** view. Two additional links to special views are also available on the top right corner: **Show recent activity** and **Show all unanswered**.

Forums are intended to be used as main generic topics on the board. We can configure the RSS feeds associated as well.

Adjusting permissions on Ploneboard for intranet use

Ploneboard defines several custom workflows assigned to its content types but they do not use the Plone 3 role schema, although we can use them as a template to create our own set of workflows suitable for our intranet.

We can find an example set of workflows for Ploneboard in the support code of this chapter. We can install them by installing a custom add-on product called `packt.p3intranets9` in our buildout, or use the supplied one. Add these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = packt.p3intranets9
...
[instance]
...
zcml = packt.p3intranets9
```

Then, rerun buildout:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Go to the Add-on Products control panel configlet and install it. A new set of workflows will be assigned to the PloneBoard content types. We can check it out on the **portal_workflow** tool in the ZMI.

For the Message Board content type, we can choose from the following states:

- **Open:** Contributors can add Forums, and editors and managers can manage them
- **Private:** Only editors and managers can access them

For the Forum content type, we can choose from the following states:

- **Require be contributor to post:** Contributors can add conversations and comments, whereas editors and managers can edit and manage them
- **Moderated forum:** Contributors can add conversations and comments, but they are moderated by users with the Reviewer role
- **Private to members only:** Only editors can add conversations and comments whereas users with other roles have no access

Conversations also have the following workflow states to control their behavior:

- **Active:** Depending on the forum workflow, the comment will be visible to users, and owner can modify them
- **Locked:** Only editors and managers can modify them
- **Pending:** Only owner can modify them
- **Rejected:** Can be modified by editors and managers, whereas only owner can view them



We can find more information about Ploneboard on the PyPI website:
<http://pypi.python.org/pypi/Products.Ploneboard>.

Polls and surveys

There are several use cases in which we need to ask a single question or a very specific format of form. They might have statistics and support views (such as portlets), and display the results in a fashionable way. Although a form generator such as PloneFormGen could do the job of gathering and storing data, the results are not formatted and ready for publication.

There are some products that do the right work when publishing the results, but do not have as many features as PloneFormGen. We will cover two of them: **PlonePopoll** and **PloneSurvey**. Although there was not much activity on the development of these products, they are still very useful.

PlonePopoll

This is a very simple tool to make available a single question with a set of predefined choices. The poll is meant to be displayed on a Plone portlet. We can place the PlonePopoll portlet anywhere on the site, as any other portlet. The results are displayed as a bar chart on the same portlet or in the view of the content object.

Installation

Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = Products.PlonePopoll
```

Then, rerun buildout:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Go to the Add-on Products control panel configlet and install it.

How it works

PlonePopoll has only one custom content type – Poll. This content type can be added anywhere in our site. We should define how the results should be shown. There are two options – before the user answers the question, or only after it.

While adding the portlet in a folder, we should configure how the portlet will show the available polls. There are some choices:

- Hidden: The portlet remains hidden from all users

- Newest poll: The portlet shows all the polls in the site, reverse ordered by date
- First poll in branch: When branch is meant to be the current folder
- First poll in branch and sub branches: It displays the results of the first poll it finds in a folder branch and its sub branches.
- By specifying a concrete poll: It displays the result of a poll selected by the user.

We can configure the number of polls to be shown in the portlet as well.

The vote gets registered and tied to the user name. Although we can vote again, our vote will replace the former one. In case of an anonymous vote, PlonePopoll registers the vote using cookies.

Adjusting default permissions for polls

We will need to adjust default permissions for polls in order to make them compatible with Plone 3 role schema. Following are the permissions that need to be adjusted:

- Popoll: Add polls: Adds Contributor and Manager
- Popoll: Edit polls: Adds Editor, Manager, and Owner
- Popoll: Vote: Adds Contributor, Editor, Manager, and Owner

These role mappings will be installed if we install the product `packt.p3intranets9` included in the support code of this chapter. We can find the complete Generic Setup `rolemap.xml` profile in the `profiles/default` folder of this product. More information on Generic Setup profiles is available in *Chapter 10, Basic Product Development*.

Taking advantage of local roles with polls

We can use local roles to show polls only to a selected number of users, based on their rights. Take advantage of these roles to create private polls.



We can find more information about PlonePopoll on the PyPI website:
<http://pypi.python.org/pypi/Products.PlonePopoll>.

Plone Survey

Plone Survey is a powerful product written to collect data from people, such as feedback on a course, simple data collection, and so on. It has support for multiple choice and free answer questions. It also provides several views for displaying the results in several formats such as HTML, CSV, bar charts, and so on. It's also possible to configure the survey behavior by defining branches depending on the user input.

Installation

Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = Products.PloneSurvey
```

Then, rerun buildout:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Go to the Add-on Products control panel configlet and install it.

How it works

PloneSurvey provides the following content types:

- Survey: A folderish object that may contain survey fields.
- Survey matrix: Useful when we have several questions that share the same possible answers.
- Survey select question: A single question with several possible answers displayed as a selection widget.
- Survey text question: A single question with a single question for which the answer should be in the form of text.
- Sub survey: We can define a sub survey which in turn can contain more survey fields. Each sub survey, will be shown as a different page with a next/previous control.

Once the survey content type is created, it's empty, and we can add survey fields to it. A logged in user can complete the survey or save it for completing it later. We can specify several attributes of the survey as well, such as:

- Thank you message text: Shown when the user submits the results
- Saved message text: Shown if the user saves the results for later completion of the survey
- Exit URL: If we want the user to be redirected to a specific URL on completion
- Confidential: Marks the results as confidential; the user remains hidden
- Allow anonymous: Allows unauthenticated users to answer the survey
- Allow save: It allows logged in users to save the survey for finishing it later
- Survey notification e-mail address: It will send a notification each time a survey is completed
- Survey notification method: Brings about survey notification

Adjusting default permissions for surveys

We will need to adjust default permissions for surveys in order to make them compatible with Plone 3 role schema. These are the permissions that need to be adjusted:

- PloneSurvey: Reset Own Responses: Adds Contributor, Editor, Manager, and Owner
- PloneSurvey: View Survey Results: Adds Editor, Manager, and Owner

These role mappings will be installed if we install the product `packt.p3intranets9` included in the support code of this chapter. We can find the complete Generic Setup `rolemap.xml` profile in the `profiles/default` folder of this product. More information on Generic Setup profiles is available in *Chapter 10, Basic Product Development*.



We can find more information about PloneSurvey on the PyPI website: <http://pypi.python.org/pypi/Products.PloneSurvey>.

Document files management

Eventually, our intranet will contain a lot of files. Hence, we should have the right tools to ease the management and consumption of files for our users. A very valuable feature that Plone provides out-of-the-box is indexing of PDF and MS Word files if a suitable support application is installed on the production server. The tools that we cover in this section are intended not only to index more file types, such as MS Office and OpenOffice file formats, but also to preview them on our Plone site without having to download and open them with a suitable desktop application.

ARFilePreview and AROfficeTransforms

These two products take care of providing the ability to index and preview a variety of file formats. These include:

- MS Office formats (DOC, XLS, PPT)
- OpenOffice formats (SXW, SXC, SXI, ODT, ODS, ODP)
- PDF
- ZIP

Installation

Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = Products.ARFilePreview
      Products.AROfficeTransforms
```

Then, rerun buildout:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Go to the Add-on Products control panel configlet and install both of them.

Features

ARFilePreview adapts the existing file content type. It provides a built-in HTML preview of the file and full text indexing without the need of any other tool. ARFilePreview uses standard Plone tools in order to do the trick. ARFilePreview is fully compliant with WebDAV, External Editor, FTP, and any third-party mass loader.

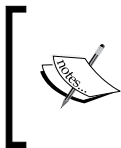
Additional software required

AROfficeTransforms contain new transforms and need additional software to be installed on the server. GNU/Linux-based servers need the following binaries:

- ppthtml
- xlhtml
- wv
- xsltproc
- unzip
- pdftohtml

For a UBUNTU-based server, we should install the packages containing these binaries, executing the following command line:

```
$ sudo apt-get install ppthtml xlhtml wv xsltproc unzip poppler-utils
```



We can find more information about these products on the PyPI website: <http://pypi.python.org/pypi/Products>. ARFilePreview and <http://pypi.python.org/pypi/Products.AROfficeTransforms>.

OpenXML

This product provides indexing and a set of icons for MS Office 2007 OpenXML-based documents.

Installation

Insert these additional lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = Products.OpenXML
```

Then, remount buildout:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Go to the Add-on Products control panel configlet and install both of them.

Dependencies

OpenXML has the following requirements:

- openxmllib 1.0.0 for Python (<http://code.google.com/p/openxmllib/>)
- lxml

However, the buildout process will take care of the dependencies and will download and install them.



We can find more information about this product on the PyPI website:
<http://pypi.python.org/pypi/Products.OpenXml>.

Summary

We have exposed a suitable stack of products addressed to intranets. We have covered the most reliable, maintained, and fully-featured set of Plone's third-party add-on products.

- Calendaring and events management
- Blogs: Quills and Scrawl
- Discussion board: PloneBoard
- Form generator: PloneFormGen
- Surveys and polls: PloneSurvey and PlonePopoll
- Document management

In the next chapter, we will introduce how to create our own add-on product for our intranet and how to use the power of Generic Setup to do the initial setup.

10

Basic Product Development

We've discussed a lot about third-party products; how to install them, how to choose them, and we've even made a recompilation of the most interesting products for an intranet. Now it's time to cover how a product is made. This chapter is also meant to serve as a guide to know where to find things in any Plone product.

But first of all, let's begin by making things clear: this chapter is merely introductory, and its intention is to introduce the reader to the basics of Plone product development, its jargon, the technologies implied, and its entities. We will learn more about `GenericSetup`, a core Plone component, and it will help us accomplish simple tasks, but with valuable results.

For more complicated use cases and in-depth learning of Plone product development, I strongly recommend the Bible of Plone developers: Martin Aspeli's, *Professional Plone Development* published by Packt.

This chapter will cover the following topics:

- Making a product with PasteScript templates
- `GenericSetup` (GS)
- Cloning a content type via GS
- Using GS to configure security
- Dexterity: A new compelling way to create content types

Later on in this chapter, we will cover a new compelling way of building new content type—Dexterity. This new technology allows us to create a content type using the web content types, and choose how they will behave, and their functionality.



For all examples in this chapter, it is assumed that we have installed all the software explained in *Chapter 2, Getting Started*. If not, we will have to install PasteScript and ZopeSkel, as instructed in Chapter 2.

Building our own product

Building our own custom product for Plone is not a big deal. PasteScript will help us generate all the boilerplate needed to build a Plone product egg.

In a buildout, there is a special folder called `src`. This folder is reserved for the products eggs in development. The development eggs should be declared in our `buildout.cfg` file in order to instruct the buildout to load them each time the buildout process is launched. Hence, we will proceed to create our new product in this reserved folder. But first, we will have to decide the name and the namespace for our package.

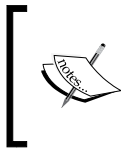
Naming our product

Eggs can share a single top-level namespace; this makes possible the distribution of multiple packages that share the same top-level namespace. This is in fact a feature of `setuptools`. For example, the packages `plone.theme` and `plone.portlets` both share the top-level "plone" namespace, but they are distributed as separate eggs. When installed, each egg's source code has its own directory (or possibly a compressed archive of that directory).

Namespaces are useful to avoid collisions and logically group related functionality, both in packages and as a general programming principle. They can be used for description, ownership, categorization, and branding. However, there are some conventions on namespaces as well in the community. For example, use `plonetheme.*` for theme eggs. There are opinions for all likings, but we will only say one thing: don't abuse the use of namespaces and use common sense.

We can do double nesting of the namespace of an egg, but it should be avoided unless there's a good reason. For example, the namespace `plone.app.*` is meant to refer to an egg that contains the code for specific functionality only reusable in Plone, while `plone.*` refers to eggs that contain core coding, and probably reusable outside the Plone world.

The use of the `Products.*` namespace can be avoided since Plone 3.3 because it includes the `z3c.autoinclude` package. It's only used for old-style products that have been eggified or upgraded to Plone 3. We can add some lines of code to our egg that makes the use of ZCML slugs unnecessary.



We can learn more about how to name our products in this entry of Martin Aspeli's blog: <http://www.martinaspeli.net/articles/the-naming-of-things-package-names-and-namespaces>.

So, for all the following examples we are going to name our product `firstproduct` and we will include it in `my` namespace.

Creating the egg

Inside our buildout folder, change the current folder to the `src` folder:

```
$ cd src
```

Let's generate our egg by typing the following command line:

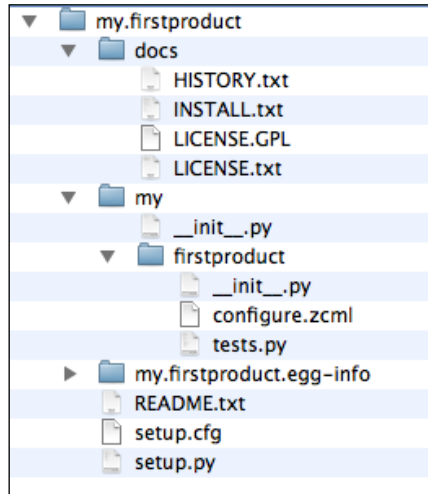
```
$ paster create -t plone my.firstproduct
```

This will instruct paster to create a new Plone product egg called `my.firstproduct` using the `plone` template, included in ZopeSkel package.

ZopeSkel asks to use the question mode during the generation of the package. The easy mode is very suitable if we don't want an advanced customization of the package. If we've already supplied the name and namespace of the package, the easy mode will create the egg correctly using the namespace specified.

Anatomy of a Plone product egg

There are some conventions and best practices in Plone (or Zope) product development. The reason is to keep products standardized and homogeneous. Following is the schema of the folders and files generated by ZopeSkel.



Egg documentation files

The egg contains three documentation files: `HISTORY.txt`, `INSTALL.txt`, and `README.txt`. The first two are present in the `docs` folder and the last file in the root of the package.

The aim of file `HISTORY.txt` is to contain all the changes made to the package throughout its life cycle. The file, `INSTALL.txt`, is used to contain the instructions for installing the package successfully.

The file `README.txt` is intended to contain all the relevant documentation information about the package. Don't forget to document your product well! This will help us and others understand the purpose, functionality, and features of our product.

`LICENSE.GPL` and `LICENSE.txt` provide the standard GNU Public License information.

Egg setup files

There are two configuration files in the root of the package: `setup.py` and `setup.cfg`. They are used to contain some important egg metadata and attributes. This is the default content of the `setup.py` file:

```
from setuptools import setup, find_packages
import os
version = '1.0'
setup(name='my.firstproduct2',
      version=version,
      description="",
      long_description=open("README.txt").read() + "\n" +
                       open(os.path.join("docs", "HISTORY.txt")).
read(),
      # Get more strings from
      # http://pypi.python.org/pypi/%3Aaction=list_classifiers
      classifiers=[
          "Framework :: Plone",
          "Programming Language :: Python",
      ],
      keywords='',
      author='',
      author_email='',
      url='http://svn.plone.org/svn/collective/',
      license='GPL',
      packages=find_packages(exclude=['ez_setup']),
      namespace_packages=['my'],
      include_package_data=True,
      zip_safe=False,
      install_requires=[
          'setuptools',
          # -*- Extra requirements: -*-
      ],
      entry_points="""
# -*- Entry points: -*-
[z3c.autoinclude.plugin]
target = plone
""",
      setup_requires=["PasteScript"],
      paster_plugins=["ZopeSkel"],
  )
```

Here we can set up the package version, description, keywords, author, URL, dependencies (`install_requires`), entry points, and other package metadata. We should fill these values carefully in case we want to release the package, for example to PyPI. We will notice that the `long_description` will be compiled by joining the `README.txt` and `HISTORY.txt` files. For these reasons, it is very important to keep these files up to date.



For more information on how to release a package to Python Package Index (PyPI), visit <http://wiki.python.org/moin/CheeseShopTutorial>.

Main product content

Under the hierarchy of folders formed by the namespace and name of our package, we will find the contents of the product. They will be a collection of code (`*.py`), Zope's configuration files (`*.zcml`), page templates (`*.pt`, `*.cpt`), browser views, skin layers, `GenericSetup` profiles, and so on. The location of all these types of files, that is composed of a Plone product, is set by convention in designated folders inside the egg package:

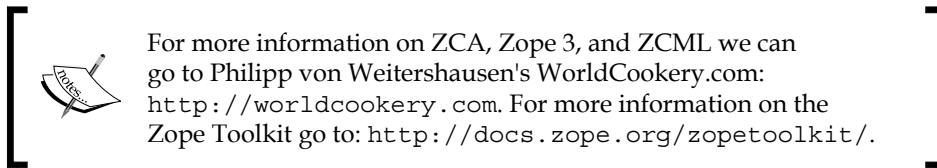
Type	Location folder	Description
Browser views and templates	browser	All the Zope browser views and the templates associated to them are stored in this folder. We also store the ZCML file with the configuration of these views here.
Zope resources	browser/stylesheets browser/images	Folders used to store Zope resources in the browser folder, separated by type.
Content types	content	This folder will hold all the code and definition of any new content type.
GenericSetup profiles	profile/default	Here we store the GS profiles that set up diverse aspects and initial configuration of our site.
Zope Skin layers	skin/*	Inside this folder, we can find Zope skin layers containing support resources for our product, if any. They may contain page templates, CSS files, and images.
Internationalization files	i18n or locales	The <code>.po</code> files containing the locales for our package.
Portlets definition	portlets	This folder will contain the configuration, templates, and code for portlets defined in the package.
Test files	tests	Contains the test code for the package.

As we've already noticed, ZopeSkel only generates the folder structure of the package and the first `configure.zcml` file. Now we have all the boilerplate in place, it's time to make it work.

ZCML configuration files

Zope Configuration Markup Language (ZCML) is used by products and modules in the **Zope Component Architecture (ZCA)** to declare components and make them available for Zope. Usually this process is called **wiring**, because it wires Python classes and page templates with names, interfaces, and layers, and gives them particular access permissions.

Although the ZCA was conceived initially for Zope 3, the Five module makes the ZCA available for Zope 2. Recently, due to the overloading of the Zope 3 term, the collection of libraries shared by Zope 2 and 3, along with some new additions to the ZCA were replaced by a more generic name – the **Zope Toolkit**.



Initially, we will only have one configuration file. This file resides in the main package folder, in our case in `my/firstproduct`, and is called `configure.zcml`. It is almost empty by default with only a few lines of boilerplate, but ready for us to complete it.

Making the product installable

Our package is innocuous for our buildout and does nothing special. Even if we declare it in our buildout, the result will be unnoticeable. Before we tell our buildout about it, let's make it available for Plone to install. We will accomplish this by adding a few lines of code to `configure.zcml`:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:genericsetup="http://namespaces.zope.org/genericsetup"
  i18n_domain="my.firstproduct">
<five:registerPackage package="" initialize=".initialize" />
<!-- Register an extension profile to make the product installable -->
  <genericsetup:registerProfile
    name="default"
```

```
    title="My first Plone product"
    description="Here goes the description of my product"
    directory="profiles/default"
    provides="Products.GenericSetup.interfaces.EXTENSION"/>
</configure>
```


All ZCML files have XML markup format and the entire declarations are inside the `configure` directive. We just have to add the `genericsetup:registerProfile` directive provided by the namespace, `genericsetup`. This will instruct `GenericSetup` to make `portal_quickinstaller` aware of our product.

If we don't want to define an `i18n` folder, we can delete the line (if it exists):

```
<i18n:registerTranslations directory="locales" />
```

Now, it's time to tell the buildout about our new product. In the `buildout.cfg` we must add these lines:

```
[buildout]
...
develop = src/my.firstproduct
...
[instance]
...
eggs = my.firstproduct
```

 Notice we should tell buildout that our product is under development and we should treat it as a source egg. We do this by including the egg path (relative to the buildout directory) in the `develop` attribute declaration. We can add as many packages as we want. It is not necessary to inform the buildout about including a ZCML slug, because we have previously instructed our package to use the `z3c.autoinclude` plugin in `setup.py`.

Next, just rerun our buildout again, and restart our instance:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Our product will be available for installation in control panel

Add-on products configlet:

Add/Remove Products

▲ Up to Site Setup

This is the Add-on Products install section, you can add and remove products in the lists below.

To make new products show up here, add them to your buildout configuration, run buildout, and restart the server process. For detailed instructions see [Installing a third party product](#) .

Products available for install

- Marshall 1.2.1**
- My first Plone product 1.0**
 Here goes the description of my product
- NuPlone 1.0b3**
 An alternative theme for Plone 3
- OpenID Authentication Support 1.1**
 Adds support for authenticating with OpenID credentials in a Plone site
[Product Description](#)
- Workflow Policy Support (CMFPlacefulWorkflow) 1.4.2**
 Add in Plone the capability to change workflow chains for types in every object.
- Working Copy Support (Iterate) 1.2.3**
 Adds working copy support (aka. in-place staging) to Plone.
[Product Description](#)

We can install it as any other add-on product for Plone. The only thing is that our product does nothing, yet.

The power of GenericSetup

Before continuing, we will have to take a short break to learn more about GenericSetup. We've only mentioned it once or twice in the book, and explained it very briefly, but in fact, it is an important component of Plone. So it's time to take care of it. It basically features import and export information from our Plone site's configuration. It uses profiles to achieve this. A **profile** is a collection of XML files, which are mainly used to provide initial settings and configuration to our site and some GenericSetup enabled add-on products.

It resides in the `portal_setup` tool in the ZMI. `GenericSetup` also features other interesting functionalities, such as the ability to perform snapshots of the current configuration of our site. We can save these snapshots to keep track of all the changes we've made to our site throughout its life cycle. We can compare two snapshots, and of course, export this information to XML files to replicate the same configuration in other sites.

The Plone site creation process itself uses `GenericSetup` to provide all initial configuration and settings to our new site. Products can also use this to configure a site or to configure some aspect of the current site. This configuration is triggered *only* when the product is installed.

Snapshots

If we access the `portal_setup` tool in the ZMI site root, we will see the default view, which doesn't hold many interesting things but the log registries of the last `portal_setup` transactions. The interesting part comes in the other tabs. Access the **Snapshots** tab, and click on the button **Create a Snapshot**:



It will create a dump of all the relevant information about our site and it will store it in the ZODB. We can access this information via the link created under **Available Snapshots**. In the background following this link, we will find the view showing all the XML files from the snapshot. Some of them are organized in folders grouped by type or functionality, for example the workflow definition folder.

Each file refers to a particular functionality, tool, or site configuration. If we click on any of them, the XML for that aspect of the site is shown. The following table is a short list with an explanation of the most important `GenericSetup` XML files:

Name of the XML file	Description
<code>actions.xml</code>	Describes all the actions stored in the <code>portal_actions</code> tool. An action is an element that performs some kind of task over a particular aspect of Plone. Some of these actions are document actions (such as print, send to, and so on), buttons, and folder options (such as cut, copy, and so on). Portal tabs are also actions.
<code>catalog.xml</code>	Defines the site catalog indexes and metadata.
<code>contentrules.xml</code>	The content rules defined in the site are described here.
<code>controlpanel.xml</code>	Holds the information about the configlets available in the site.
<code>cssregistry.xml</code> , <code>jsregistry.xml</code>	Defines the information about the site's CSS and JavaScript resource registry.
<code>factorytool.xml</code>	Describes which content types will use the factory tool.
<code>mailhost.xml</code>	Describes the site's mail setup.
<code>memberdata_properties.xml</code>	Several default properties associated to any site user are defined here.
<code>portal_languages.xml</code>	The languages supported by the site are defined here.
<code>portlets.xml</code>	Defines the portal portlets and initial assignments.
<code>properties.xml</code>	Holds the main properties and attributes of the site, such as the site's title, encoding, and so on.
<code>propertiestool.xml</code>	It describes the site's properties sheets stored in the <code>portal_properties</code> tool.
<code>rolemap.xml</code>	Defines the root folder security permissions.
<code>skins.xml</code>	We can find here the definition of the site's skin layers.
<code>types and types.xml</code>	The folder contains the definition of the site's types and the file declares them.
<code>viewlets.xml</code>	We can find the definition of the visibility of existing viewlets, its position, and declarations for new viewlets.
<code>workflows and workflows.xml</code>	The folder contains the definition of each site's defined workflow and the file enumerates them.

This dump defines the state of our site at the moment we take the snapshot. We can use the information contained here to build our own GenericSetup profiles for importing the configuration defined in them into the site. As a matter of fact, we can use a snapshot as a template for our own definition profiles. More information on this is available later in the chapter.

Importing and exporting a particular product profile

We will find import and export under their respective tabs in `portal_setup`. We can export the entire current site configuration to a `tgz` file that will contain the same XML files and content, as shown in a `portal_setup` snapshot.

We can eventually import a particular configuration from a single file or another `tgz` bundle containing GenericSetup information. However, it's easier to do it through a custom product, as we will show in the next section.

Comparing snapshots and product profiles

A comparison between snapshots and product profiles is also available. It is intended for easily detecting the differences between two configuration definitions. Let's see it in action.

Provided we've already done an initial snapshot following the previous section's examples, let's change the site title. Go to your Plone's instance ZMI root and access the **Properties** tab. Change the title's property. Then, return to the `portal_setup` tool and take another snapshot. Go to the **Comparison** tab and perform a comparison between the previous snapshot and the latter one. The following screenshot is the result; can you see the value I've set in my site title?

Contents
Profiles
Import
Export
Upgrades
Snapshots
Comparison
Manage
Security
Undo
Ownership

Generic Setup Tool at /intranet/portal_setup

Setup Tool

By selecting two snapshots (or a snapshot and a filesystem setup directory), a comparison can be made, highlighting the differences between the two configuration sets.

Configurations to compare:

snapshot-20100516124105
snapshot-20100516124822

Treat missing files as empty
 Ignore lines of whitespace

Comparison of snapshot-snapshot-20100516124105 and snapshot-snapshot-20100516124822:

```

Index: properties.xml
=====
--- properties.xml 2010/05/16 14:41:07.188 GMT+2
+++ properties.xml 2010/05/16 14:48:24.006 GMT+2
@@ -1,6 +1,6 @@
<?xml version="1.0"?>
<site>
- <property name="title">Site</property>
+ <property name="title">My corporate intranet</property>
  <property name="description"></property>
  <property name="default_page" type="string">front-page</property>
  <property name="selectable_views" type="lines">
Index: structure/.properties
=====
--- structure/.properties 2010/05/16 14:41:07.188 GMT+2
+++ structure/.properties 2010/05/16 14:48:24.006 GMT+2
@@ -1,4 +1,4 @@
 [DEFAULT]
 description =
 -title = Site
 +title = My corporate intranet

```

Importing GenericSetup profiles from a product

After this short introduction to GS, it's time to show how to work with it and get the most out of our custom product. It's very easy to use it in our products, and it gives immediate results and great value. It can help us in configuring and deploying instances repeatedly.

We can put our custom GS profiles in the folder `profiles/default` of the main product content, inside our custom product egg package. All these profiles will be loaded into the site each time the product is installed.



We must follow the same names and folder hierarchy shown in a `portal_setup` snapshot. It's recommended to take a snapshot and use it as a template of the XML profiles we want to use.

Cloning content types via GenericSetup

This is a very useful feature, making use of GS profiles to define a new portal content type by cloning an existing one, without writing any line of code.

Suppose we want to clone the default file content type and clone it into a more specific content type; for example, to store all spreadsheets documents.



Particularizing content types

The example shown is a smart move in order to improve the search for different kinds of content in our site. If our users are searching for a specific type of file, it's easier to implement a custom search for a more specific type of file than searching over a general content type. It's a very commonly used practice in types management. We can split a generalist content type, such as file into more specific ones, for example spreadsheet, word processor, or PDF files. Other content types can be split as well, such as news or events into press releases or meetings. Users find this very useful and it helps us build a more humane CMS.

The method is very simple. Generally, using snapshot information as a template for our product profiles is a good practice. We should create a folder called `profiles` in the main product folder and another one called `default` inside it.

```
$ cd src
$ cd my.firstproduct/my/firstproduct
$ mkdir profiles
$ cd profiles
$ mkdir default
$ cd default
```

In order to define a new content type, we must provide a `types.xml` file containing the declaration of the new content type; and a folder called `types`, inside which you should create a file with the name of the new content type. Let's say that the new content type name will be `spreadsheet`. Then the file must be called `spreadsheet.xml`. Use our favorite text editor to edit the `types.xml` file:

```
$ vi types.xml
```

The `types.xml` file will contain this code, where we declare the existence of the new content type:

```
<?xml version="1.0"?>
<object name="portal_types" meta_type="Plone Types Tool">
  <object name="Spreadsheet"
    meta_type="Factory-based Type Information with dynamic views"/>
</object>
```

Then create the folder:

```
$ mkdir types
$ cd types
```

And edit the `spreadsheet.xml` file:

```
$ vi spreadsheet.xml
```

The `spreadsheet.xml` will contain this code, where we define the attributes of the new content type.

```
<?xml version="1.0"?>
<object name="Spreadsheet"
  meta_type="Factory-based Type Information with dynamic views"
  i18n:domain="plone" xmlns:i18n="http://xml.zope.org/namespaces/
i18n">
  <property name="title" i18n:translate="">Spreadsheet</property>
  <property name="description"
    i18n:translate="">An external spreadsheet file uploaded to the
site.</property>
  <property name="content_icon">file_icon.gif</property>
  <property name="content_meta_type">ATFile</property>
  <property name="product">ATContentTypes</property>
  <property name="factory">addATFile</property>
  <property name="immediate_view">file_view</property>
  <property name="global_allow">True</property>
  <property name="filter_content_types">True</property>
  <property name="allowed_content_types"/>
  <property name="allow_discussion">False</property>
  <property name="default_view">file_view</property>
  <property name="view_methods">
    <element value="file_view"/>
  </property>
  <property name="default_view_fallback">False</property>
  <alias from="(Default)" to="index_html"/>
  <alias from="edit" to="atct_edit"/>
```

```
<alias from="sharing" to="@@sharing"/>
<alias from="view" to="(selected layout)"/>
<action title="View" action_id="view" category="object" condition_
expr=""
    url_expr="string:${object_url}/view" visible="True">
    <permission value="View"/>
</action>
<action title="Edit" action_id="edit" category="object"
    condition_expr="not:object/@@plone_lock_info/is_locked_for_
current_user|python:True"
    url_expr="string:${object_url}/edit" visible="True">
    <permission value="Modify portal content"/>
</action>
<action title="References" action_id="references" category="object"
    condition_expr="object/archetype_tool/has_graphviz"
    url_expr="string:${object_url}/reference_graph" visible="True">
    <permission value="Modify portal content"/>
    <permission value="Review portal content"/>
</action>
<action title="Download" action_id="download" category="object"
    condition_expr="member" url_expr="string:${object_url}/download"
    visible="False">
    <permission value="View"/>
</action>
<action title="External Edit" action_id="external_edit"
category="object"
    condition_expr="object/externalEditorEnabled"
    url_expr="string:${object_url}/external_edit" visible="False">
    <permission value="Modify portal content"/>
</action>
</object>
```

This is the same code used to define a file content type, but with the slightest difference indicated by the highlighted code, where you should customize these properties with the cloned type ones. We can also modify other properties, such as the `content_icon` file that will define the icon used in the **Add new...** menu.

We must reinstall the product in the Plone control panel, under the **Add-on products** configlet to apply the changes. Now, we should see the new content type available to use in the **Add new...** menu.

We can make the new content type factory tool aware. This tool enables a temporary stage for a content type at the moment of its creation. If the creation of the content type is not completed, then the object is discarded. We can enable the factory tool for our new content type by declaring it in the `factorytool.xml` file:

```
<?xml version="1.0"?>
<object name="portal_factory" meta_type="Plone Factory Tool">
  <factorytypes>
    <type portal_type="Spreadsheet"/>
  </factorytypes>
</object>
```

Using a product to configure security

As pointed out in *Chapter 6, Managing Workflows*, we can use GS to configure two security aspects of our intranet: default role map assignment to permissions and configure existing workflows or define new ones.

Defining role map assignment to permissions

We can define role map assignment to permissions and define new roles as well with a GS profile. The file used to do this task is called `rolemap.xml`. This is an example of a `rolemap.xml` file, if we need to define new roles for our site:

```
<?xml version="1.0"?>
<rolemap>
  <roles>
    <role name="CanDelegateRoles"/>
  </roles>
  ...
```

To map this new role, we've just created a permission:

```
...
<permissions>
  <permission name="Sharing page: Delegate roles"
    acquire="True">
    <role name="CanDelegateRoles"/>
  </permission>
</permissions>
</rolemap>
```

Use the `role` definition inside the `roles` directive to define new roles for our site. We can add as many roles as we want, but as a rule of thumb, just create those roles which are really needed and try to make them as descriptive as possible.

Creating new workflows or modifying existing ones

Creating new workflows via GS is also possible. We can modify existing ones as well. There are several files and folders involved with workflows inside a GS profile, such as the `workflows.xml` file and the `workflows` folder.

The file, `workflows.xml`, defines which workflows will be managed by this GS profile. Let's use the `myintranet_workflow` created in *Chapter 7, Securing our Intranet* as an example. We can find it in the support code of Chapter 7. We can define it in the file `workflows.xml` in the following way:

```
<?xml version="1.0"?>
<object name="portal_workflow" meta_type="Plone Workflow Tool">
  <object name="myintranet_workflow" meta_type="Workflow"/>
</object>
```

Notice that in this file we should only declare which workflows will manage this GS profile. The actual definition for each of the declared workflows should be stored inside a folder with the same name of the workflow located inside the GS profile `workflows` folder. The file containing the workflow information should be stored in a file named `definition.xml`. For the `myintranet_workflow` example, we will find a `myintranet_workflow` folder inside the GS profile `workflows` folder containing the file `definition.xml` with the following contents:

```
<?xml version="1.0"?>
<dc-workflow workflow_id="myintranet_workflow"
  title="My example intranet workflow"
  description="A workflow having three states: private,
  draft and intranet."
  state_variable="review_state"
  initial_state="visible">
  <permission>Access contents information</permission>
  <permission>List folder contents</permission>
```

```

<permission>Modify portal content</permission>
<permission>View</permission>
<state state_id="intranet" title="Intranet">
  <description>Intranet state.</description>
  <exit-transition transition_id="retract"/>
</state>
<state state_id="private" title="Private">
  <description>Can only be seen and edited by the owner.
</description>
...

```



The contents of this file have been cropped because it's very extensive to cover it in this book. However, we can find it in the folder `my.firstproduct/my/firstproduct/profiles/default/workflows/myintranet_workflow/definition.xml` in the support code of *Chapter 7, Securing our Intranet*.

The structure of this file is complex and large because of the number of elements and variables involved in a workflow: states, transitions, scripts, variables, and so on, should be defined and configured. Although we can modify it directly, it's more common to use the tools provided by Plone to define or modify a workflow and then export it.



We can take advantage of GS snapshot and export features to build our own GS workflow definition for our custom product. Just create and adjust our custom workflow with the tools and methods provided in *Chapter 6, Managing Workflows* and then export it. Then add the definitions to our custom product. Once done, we will have our custom workflow ready to use in any of our sites by installing our custom product in them.

We can modify an existing workflow as well by redefining it using GS, in the same way we create a new workflow.

Dexterity

This is another word to add to the already long list of Plone's jargon. But what an awesome and extraordinary one! Dexterity is a content type framework for CMF applications. Dexterity is said to be the successor of Archetypes, the current content type framework used by Plone, as it is more lightweight and modular. Following are the Dexterity key features:

- We can create a content type entirely through the Web without any previous programming knowledge
- We can create content types easily and quickly using filesystem code
- We can assign general behaviors to a content type, such as title to ID naming, support for locking and versioning, add sets of metadata and add multilingual extensions
- We can augment content types designed through the Web or filesystem with adapters, event handlers, and other Python code written on the filesystem
- We can easily package and distribute content types designed through the Web, filesystem, or a combination of the two

Based on Zope Toolkit technologies, Dexterity is designed to be small, and easy to learn. As it reuses existing concepts, it doesn't rely on generated code, and tries to avoid things that happen *automagically*.

An intranet can gain a lot of profit from a tool like this. Imagine people that need a particular content type to store some information. The normal process would be to ask you or your IT department for a new content type. With Dexterity, it would be very easy for a non-technical user to build an entirely new content type, add the fields and behaviors he or she may need, and make it available to all intranet users.

However, at this point, Dexterity is still in alpha stage, and is not ready for production, although we can use it at our own risk. It's expected that the first beta version would be available when Plone 4 is ready.



We can find an excellent tutorial written by Martin Aspeli on plone.org. He is leading the Dexterity project, so it guarantees solid code and good work. We can find it at <http://plone.org/products/dexterity/documentation> and <http://plone.org/products/dexterity/documentation/manual/developer-manual>.

Summary

This chapter is a brief introduction to product development. It gives an in-depth knowledge, aims at introducing the reader to the basics, and addresses him to the right places and texts to learn more about the matter.

It covers the following topics:

- Making a product with PasteScript templates
- GenericSetup (GS)
- Cloning a content type via GS
- Using GS to configure security
- Dexterity: a new, compelling way to create content types

By now, we should be used to development jargon, the technologies implied, and also how an add-on product is structured. We've learnt how to take advantage of GenericSetup and use it to configure our site security. We should also know where to get more information about this subject.

In the next chapter, we will learn about how to theme our intranet.

11

Content Rules, Syndication, and Advanced Features

Some of Plone's advanced features at user level are worth having their own section. All of them have a direct impact on how our users use the intranet, and most importantly, they are the catalyst to an alive and more dynamic intranet. A dynamic intranet is in constant change and users update its contents frequently. In this chapter, we will cover the following topics:

- **Content rules:** They will allow us to define a set of actions and tasks triggered when some event happens in our site, or in a folder tree. Both the actions and events are user configurable and help us make our site dynamic.
- **Syndication:** This is often very important in order to keep our users posted when something changes in our intranet. Not only collections are syndication aware, we can also make any folder in our site export the objects it contains as an RSS feed.
- **Versioning:** This is another notable Plone feature and very useful in an intranet scenario. In few words, our users will love it.
- **WebDAV access:** WebDAV access to content, along with external editing, will enable communication between our user's desktop and the intranet, taking our user's productivity to its maximum.
- **External editing:** This feature will allow us to edit any file content type with the suitable desktop application and save it on the fly.

Content rules

Plone features a usability layer around Zope's event system, allowing plain users to create rules tied to the most used event handlers. These rules are composed of tasks that get triggered when an event is raised in our site. Content rules are defined site-wide in the **Content rules** configlet, and they are available for use in any folderish object in our site. Once the rule is created, it can be locally assigned to any folder object in the site.

Rules play a very important role in intranets. We can use them as a mechanism for notification, and they also help in adding dynamism to our intranet. One of the most demanded features in an intranet is the ability to be aware when content is added, changed, or even deleted. The notification of this change to the users can be achieved via content rules assigned strategically, or by user demand in any folder or intranet application, such as forums or in a blog.

We can use content types to help us model some of our corporate processes or daily tasks. Move or copy objects to other folders (done by users), just in case some of our processes require this kind of an action. We can find other interesting uses of content rules in our intranet, such as executing an action when a state transition is triggered.

All these actions can be carried out programmatically, but the power of content rules lie in that they can be executed thorough the Plone UI and by any experienced user.

We can access the manage rules form via the **Rules** tab in any folder. If we don't have any rules created, the form will address us to create them in the content rules configlet. This control panel configlet will aid us to create and manage content rules of our site:

Content rules

▲ Up to Site Setup

Use the form below to define, change or remove content rules. Rules will automatically perform actions on content when certain triggers take place. After defining rules, you may want to go to a folder to assign them, using the "rules" item in the actions menu.

Global settings

The following settings affect rules globally.

Enable globally

Whether or not content rules should be enabled globally. If this is deselected, no rules will be executed anywhere in the portal.

Content rules

Overview of the content rules that can be assigned to folders and other types of containers in the portal.

Show

The form is divided into two parts. The first is dedicated to global settings applied to all rules. In this version, there is only one setting in this category to enable and disable the rules in the whole site. If deselected, the whole rule system is disabled and no rules will be executed in the site.

The other part of the form is reserved for the rule management interface. Here we can find the already created rules, manage them, and create new ones. We can display them by type using the selector on the right.

Adding a new rule

Click on the **Add content rule** button. It will open a new form with the following fields:

- Title: Title of the rule.
- Description: Summary of the rule.
- Triggering event: Starts the execution of the rule.
- Enabled: Whether or not this rule is enabled.
- Stop executing rules: Defines if the engine should continue the execution of other rules. It is useful if we assign several rules to a container and the execution of a particular rule excludes any other rule execution.

By default, these are the available events:

- Object added to this container
- Object modified
- Object removed from this container
- Workflow state changed

Add Rule

Add a new rule. Once complete, you can manage the rule's actions and conditions separately.

Configure rule

Title ■
The title of the rule

Description
A summary of the rule

Triggering event ■
The event that can trigger this rule

Object added to this container

Enabled
Whether or not the rule is currently enabled

Stop executing rules
Whether or not execution of further rules should stop after this rule is executed

After creating one rule at least, the configlet will let us manage the existing rules, allowing us to perform the standard edit, delete, enable, and disable actions. But this is only the first step. We've created the rule and assigned an event to it. Now it's time to configure the task, which the rule will perform. There are two items to configure—conditions and actions.

We can add as many conditions as we want to, and modify the order in which they can be applied. We can add the following types of conditions:

- Content type: Apply the rule only if an object of this type has triggered the event
- File extension: Execute the action only if a file content type that has this extension has triggered the event
- Workflow state: Apply only if a content type in the workflow state specified has triggered the event
- User's group: Execute only if a user member of a specific group triggers the event
- User's role: Same as User's group, but by a user having a specific role in that context

The actions that a rule can execute are limited but they cover the most useful use cases:

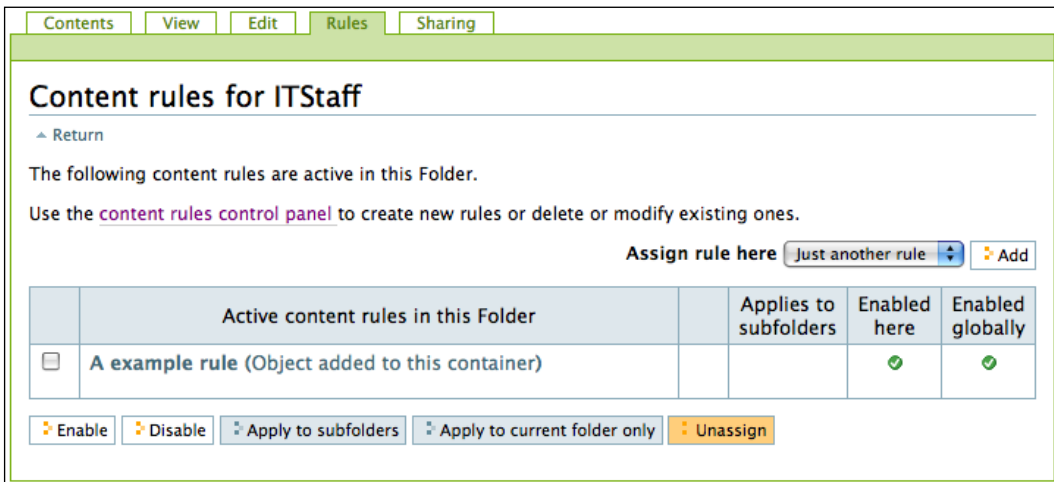
- Logger: Output a message to the message system log
- Notify user: Notify the user via a status message
- Copy to folder: The object that triggers the event is copied to the specified folder
- Move to folder: The object that triggers the event is moved to the specified folder
- Delete object: The object that triggers the event is deleted
- Transition workflow state: An attempt to change workflow of the object that triggers the event via the specified transition
- Send e-mail: Send e-mail to a specific user



By default, only managers can define and apply new content rules, but we can allow more user roles to access their creation.

Assigning rules to folderish objects

Once the rule is created, we can assign them to any of Plone's folderish content types. Just go to any folderish object and click on the **Rules** tab.



Just use the drop-down box **Assign rule here** to choose from the available rules and click on **Add**. We can review what rules are assigned in this container and manage them as well. We can enable, disable, and choose whether to apply them to subfolders or only to current folders, and of course, unassign them.

Making any content type rule aware

All folderish default content types of Plone are content rule aware. However, not all third-party content types are content rule aware. This is because either they are old or simply do not enable this feature in the content type declaration.

In the case of third-party content types, which are not content rule aware, we can enable their awareness by following these instructions: Add an object of the desired content type anywhere in our site, if we haven't created it yet. Find it in the ZMI and access the **Interfaces** tab. Once there, find the interface `plone.contentrules.engine.interfaces.IRuleAssignable` in the **Available marker interfaces** fieldset. Select it and click on the **Add** button. By doing so, we are assigning an additional marker interface to that content type, which will enable (*mark*) this instance of the content type (that is, make it aware of the content rule). From this moment onwards, the selected object will have available the **Rules** tab, and in consequence, we can assign rules to it.


Syndication

Plone has always paid special attention to syndication, making its folderish content types syndicable. As we have seen in *Chapter 8, Using Content Type Effectively*, collections export their contents automatically in a view that all collections have – RSS view. But we can also enable syndication for single folders on our site.

Using RSS feeds in our intranet is the recommended approach for keeping our users posted about the changes in syndicated folders, if they are collections or plain folders.

Enabling folder syndication

For enabling syndication for a particular folder, we need to access the view, `synPropertiesForm`, from the folder we want to be syndicable. For example, if we want to access this view in the `ITStaff` folder, we should browse the URL:
<http://localhost:8080/intranet/ITStaff/synPropertiesForm>

 This view is hidden by default, although we can make it visible in order to allow users to enable folder syndication by themselves. We can make it visible by accessing the `portal_actions` tool in the ZMI. Go to the object action category and choose syndication. Then just make this action visible by enabling the `visible` attribute and choose who will be able to access this view by selecting the item permissions in the **Permissions** selection box.

Once in the `synPropertiesForm` form, we should click on the **Enable syndication** button. Then another form is shown to allow us to configure how the publication of the feed will be performed. Following are the syndication details available:

- Update period: How often the feed will be updated
- Update frequency: How many times the update will occur inside the period specified in the previous field

- Update base: When the update will take place
- Maximum items: How many items the feed will show

Contents View Edit Rules Sharing

Info Syndication enabled

Syndication Properties

Syndication enables you to syndicate this folder so it can be synchronized from other web sites.

Disable Syndication

Channel Title
ITStaff

Channel Description

Syndication Details

Update Period
Controls how often the channel is updated.
Daily

Update Frequency
Controls the frequency of the updates. For example, if you want it to update twice a week, select "weekly" above, and "2" here.
1

Update Base
This is the date the updater starts counting from. So if you want to update weekly every Tuesday, make sure this starts on a Tuesday.
2010 / February / 21 11 : 55 AM

Maximum Items
Maximum number of items that will be syndicated.
15

Save

Accessing a secure RSS feed

Syndication was conceived to access information from public resources. Inside an intranet, it will be very common that the folder we want to enable for syndication will be not published, and in consequence, the feed associated will be private. The problem is that there are few feed readers that support feed authentication and even using them. We will have to enable HTTP authentication in our site's PAS configuration, which is not recommended. So we propose two workarounds.

We can use a feed enabled browser to browse our intranet and our feeds as well. With this approach, if we are logged in, then we will have access to authenticated feeds. Firefox and Internet Explorer already have this feature.

The second approach is to have a special workflow state for the syndicated folders inside our site for being accessible without authentication as anonymous users. Obviously this workaround will make the folder content visible to anonymous users, and it's not an option when privacy of the contained information is a must.

Versioning

Versioning has been around since Plone 3. It keeps track of all the changes made to an object by saving them as revisions. These changes can be compared; we can undo the changes made for a specific version and switch any previous version with the current one.

Nowadays it's hard to conceive a CMS without a version system. Versioning is more a requirement than a feature.

This feature used to live on an object tab called **History**, but since Plone 3.3 it was moved to the **History** collapsible menu at the bottom of the standard view. This is how it looks; a page with several changes and the **History** menu expanded:

The screenshot shows a web page interface for a document titled "Just a page". At the top, there are tabs for "View", "Edit", and "Sharing". To the right, there are "Actions" and "State: Private" dropdown menus. The main content area contains the following text:

Just a page
 by [admin](#) — last modified Feb 21, 2010 05:30 PM
Description of the page. And now, I'm completing the description.
 The body here!
 I'm adding a line below the original one.
 And yet another one!

Below the text is an expanded "History" menu. It lists four revisions, each with a "Revert to this revision" button and a "View this revision · Compare with current revision" link. The revisions are:

- Edited by [admin](#) on Feb 21, 2010 05:30 PM
- Edited by [admin](#) on Feb 21, 2010 05:29 PM
- Edited by [admin](#) on Feb 21, 2010 05:29 PM
- Edited by [admin](#) on Feb 21, 2010 05:24 PM

At the bottom of the page, there are links for "Send this" and "Print this".

The **Change note** field of any content type is a part of the versioning system and allows us to add a comment to any version before we submit the changes to the server, creating a new version. This comment is stored with the version and is useful as a description of the changes made to the object.

We can view what the page looks like in each version using the **View this revision** link. We can compare each of the previous versions with the current one, and comparison can be made between the previous versions as well. We can also revert to the version we desire by clicking on the corresponding button.

The comparisons are made using a form that shows us the differences between versions inline or as HTML code. We can also change the version number using the **Versions** selector. The differences of the object's metadata can be seen as well.

Changing versioning policy

We can change versioning policy by using Plone's control panel **Types** configlet. If we select one particular content type, we will find the **Versioning policy** drop-down box. There are three settings: automatic, manual, and no versioning.

There are several ZMI tools implied in the versioning system as well. It's interesting to show the functionality of two of them: `portal_historiesstorage` and `portal_purgepolicy`.

Inside the first one, we can find information about storage statistics from current versions in our site. We can check the following:

- Number of histories
- Number of versions
- Average versions per history

We can see the most versioned objects and their location too.

The number of versions stored is infinite by default. However, we can change the behavior of versioning from `portal_purgepolicy` by changing the default infinite value (-1) of the **maximum number of versions to keep in the storage** attribute to the desired value.



Recommended number of stored versions

It's expected to have a high profile of changes and modifications in a moderately crowded intranet. It's advisable to keep the number of versions to a sustainable value because it's related to the space occupied by the object in the database. Every version we create of any object is stored in the database and it never gets purged. Usually keeping ten versions for all objects is a good practice and enough for almost any use case.

WebDAV

Web-based Distributed Authoring and Versioning, or WebDAV, is a set of extensions to the Hypertext Transfer Protocol (HTTP) that allows computer-users to edit and manage files collaboratively on remote World Wide Web servers. RFC 4918 defines the extensions. It allows users to create, change, and move documents on a remote server (typically a web server or "web share"). WebDAV allows us to interact with our web content, as it was files and folders on our computer. We can literally mount a WebDAV folder as a folder on our computer's filesystem.

Nowadays, every major operating system has a client implementation of WebDAV and there are many third-party clients we can use. Windows Explorer, Nautilus, and Konqueror for Linux, and Finder for Mac OS X have a WebDAV client built in.



The implementation of WebDAV clients is not homogeneous out there. Every vendor has their own implementation and may slightly differ from others. These few differences can cause the interaction with our information to fail, so be careful about it. A test with our preferred client on the most common activities when dealing with WebDAV (copy, move, delete, rename) is highly recommended.

Any default, out-of-the-box Plone object is WebDAV aware and is accessible using its URL. The idea is that we can mount a WebDAV resource from any of these clients by using the object URL.

This is the complete set of instructions for Mac OS X:

1. Open **Finder**, select **Go** menu, and then **Connect to server...** or press *Command + K*.
2. Specify the URL of the resource in the **Server address** field, for example, our site's root: `http://localhost:8080/intranet` and then click on **Connect**. We should be prompted for authentication.

After a few seconds, we will be able to see the entire site's structure as filesystem resources, via WebDAV:

Name	Date Modified	Size	Kind
▶ folder	7 February 2010, 19:13	--	Folder
▶ folder	7 February 2010, 19:13	--	Folder
■ caching_policy_manager	7 February 2010, 19:13	Zero KB	Unix ...ble File
■ content_type_registry	7 February 2010, 19:13	Zero KB	Unix ...ble File
■ error_log	7 February 2010, 19:13	Zero KB	Unix ...ble File
▶ folder	7 February 2010, 19:13	--	Folder
■ front-page	Today, 17:23	5 KB	Unix ...ble File
■ HTTPCache	7 February 2010, 19:13	Zero KB	Unix ...ble File
▶ folder	Today, 17:07	--	Folder
■ just-a-page	Today, 17:30	1 KB	Unix ...ble File
■ kupu_library_tool	7 February 2010, 19:13	Zero KB	Unix ...ble File
■ MailHost	7 February 2010, 19:13	Zero KB	Unix ...ble File
▶ folder	7 February 2010, 19:13	--	Folder
▶ folder	7 February 2010, 19:13	--	Folder
▶ folder	7 February 2010, 19:13	--	Folder
■ plone_utils	7 February 2010, 19:13	Zero KB	Unix ...ble File
■ portal_actionicons	7 February 2010, 19:13	Zero KB	Unix ...ble File
▶ folder	7 February 2010, 19:13	--	Folder
■ portal_archivist	7 February 2010, 19:13	Zero KB	Unix ...ble File
■ portal_atct	7 February 2010, 19:13	Zero KB	Unix ...ble File
■ portal_calendar	7 February 2010, 19:13	Zero KB	Unix ...ble File
▶ folder	7 February 2010, 19:13	--	Folder
▶ folder	7 February 2010, 19:13	--	Folder
■ portal_css	13 February 2010, 12:39	Zero KB	Unix ...ble File
■ portal_diff	7 February 2010, 19:13	Zero KB	Unix ...ble File
■ portal_discussion	7 February 2010, 19:13	Zero KB	Unix ...ble File
■ portal_factory	7 February 2010, 19:13	Zero KB	Unix ...ble File
■ portal_form_controller	7 February 2010, 19:13	Zero KB	--
■ portal_groupdata	7 February 2010, 19:13	Zero KB	--
■ portal_groups	7 February 2010, 19:13	Zero KB	--
■ portal_historiesstorage	Today, 17:03	Zero KB	--
■ portal_historyidhandler	7 February 2010, 19:13	Zero KB	--
■ portal_interface	7 February 2010, 19:13	Zero KB	--
■ portal_javascripts	7 February 2010, 19:13	Zero KB	--
■ portal_kss	7 February 2010, 19:13	Zero KB	--
■ portal_languages	19 February 2010, 10:20	Zero KB	--
■ portal_memberdata	7 February 2010, 19:13	Zero KB	--
▶ folder	7 February 2010, 19:13	--	--
▶ folder	7 February 2010, 19:13	--	--
■ portal_migration	7 February 2010, 19:13	Zero KB	--
▶ folder	7 February 2010, 19:13	--	--
■ portal_password_reset	7 February 2010, 19:13	Zero KB	--
▶ folder	7 February 2010, 19:13	--	--

We can perform the same operations on displayed items as they were files or folders in our filesystem. Copy, move, rename, and so on. If we have logged in as a manager, we should be able to see the content and site's tools as well. If logged in as a plain user, we will see the objects we are allowed to see depending on our permission rights.

The advantage in an intranet scenario is obvious. Having desktop client interaction with our intranet is a valuable companion and an inestimable feature. Our users can do massive file uploads, move and rename content, and even copy entire folders from the site right to their desktops.

Managing WebDAV access permissions

There are some permissions involved with WebDAV operations:

- WebDAV access
- Manage WebDAV Locks
- WebDAV Lock items
- WebDAV Unlock items

Use them to restrict or allow WebDAV access to our intranet resources. By default, only managers can access via WebDAV.

We can assign the desired roles to these permissions with a custom workflow, or set them globally by setting the proper role mapping in the `rolemap.xml` file of a GenericSetup profile in a custom add-on product.

External editing

When our users upload files to the intranet, eventually our users may want to modify these files. The standard procedure will be simple but tedious: download it, modify it locally with the associated software (let's say OpenOffice), save it locally, and upload again (modifying the original file object type), and save it.

Plone features the external edition to overcome all that. With the help of an external desktop application (External Editor), we can modify and save the file on the fly. This little Python application (of course) does all the dirty work for us. It opens the file via WebDAV, and launches the associated application in our computer with it. We can modify it, and when we save the file, automatically save it via WebDAV, modifying the original file on the server.

External Editor also manages a WebDAV lock, so nobody else could modify the object while we are editing it. When we close the application, the file is saved via WebDAV and the lock is released.

Since External Editor is Python-based, it is multiplatform, and can be run on any major platform. We can find the script (`zopeedit.py`) and a Windows packaged version in: <http://plone.org/products/zope-externaleditor-client>.

Installing the External Editor

Getting this tool installed in our computer involves two things: installing a Python script that takes care of the edition process, and the bindings with our default browser that instructs it to launch the script for a special MIME type (`application/x-zope-edit`).

Windows

Installing it in Windows is very straightforward: just run the `.exe` provided and External Editor will install it in our computer and will configure our default browser bindings.

Linux

Installing it in a Linux machine is more manual because the packages included in the official distros tend to be outdated. In case of UBUNTU, we can try this recipe as there is a `.deb` software package in the UBUNTU repositories, but it's not updated. However, we are going to use it as a base, as it will configure our browser's MIME types correctly:

```
$ sudo apt-get install zopeedit
```

Download the updated `zopeedit.py` file from the URL:

```
$ wget http://plone.org/products/zope-externaleditor-client/  
releases/0.9.11/zopeedit.py
```

Overwrite the original one installed by the `.deb` package by copying the updated version we've just downloaded from our download folder to this path:

```
$ sudo cp zopeedit.py /usr/bin/zopeedit
```

After this, we will have the script updated and a ready-to-use External Editor when we enable it on our profile.



We should probably check out plone.org for the latest version of External Editor. At the time of writing, 0.9.11 was the last version available.

Mac OS X

We need to download the External Editor packaging for OSX from plone.org:
<http://plone.org/products/zopeeditmanager/releases/0.9.7/osx-installer>. It's a Cocoa-based application for Mac OS X that implements an External Editor called ZopeEditManager.

Unzip the downloaded file (`osx-installer.zip`) and copy the application, ZopeEditManager, to our `/Applications` folder. Launch it like any other application. By default, it shows a window with the list of all the files being edited. ZopeEditManager provides a GUI Preferences panel. Just choose **Preferences...** from the ZopeEditManager menu. In this panel, we can adjust the helper applications, WebDAV properties, and other configuration.

Enabling external editing

Any user can enable external editing in his site profile. We can access it by clicking on our name's link on the personal toolbar and then clicking on **Personal profile**, or accessing the **personalize_form** view.

Mark the **Enable external editing** checkbox to enable external editing for our user and then save. When checked, a pencil icon will be visible on each page that allows us to edit content using an External Editor. When clicked, External Editor will launch the defined helper application with the specified file.

Modifying helper software

There are a number of MIME types associated with External Editor. However, we can change it and specify our preferred application to use with external edition. There is a configuration file located in our home called `.zope-external-edit`, in case of a Linux box, or in our `Documents` folder, in case of a Windows computer. We can adjust helper applications using the **Preferences...** panel in ZopeEditManager for Mac OS X.

Summary

In this chapter, we've learnt about the following advanced content features and how to take advantage of them.

- Content rules
- Syndication
- Versioning
- WebDAV access
- External edition

Don't hesitate to teach our intranet users about them; they will benefit a lot from it. The content in an intranet is not meant to be static; it should be dynamic as our users work with it and share information with each other. These tools will help them be more productive and efficient in this task.

The next chapter will cover how to theme our intranet.

12

Theming our Intranet

Theming – what a huge subject in Plone!

Eventually, we will need to customize the look and feel of our intranet by branding our site, such as customizing the logo, or doing some styling to match our company's design style-guide.

There are a lot of technologies, techniques, and tools involved in Plone theming: HTML, CSS, JavaScript, Zope's page **Template Attribute Language** (TAL), Plone skin layers, Zope3-style viewlets and views, resource registries, and so on.

Don't worry if you are not familiar with any of these. Although it is totally out of the scope of this chapter to cover all these subjects, we will try to synthesize them all and cover the most important ones, focusing on those things that will provide immediate value for us and our intranet. In this chapter, we will learn:

- How Plone renders the site
- Key Plone theming technologies
- Helper tools to implement theming easily
- Useful examples and recipes



For some sections of this chapter, a basic knowledge of HTML and CSS will be necessary.

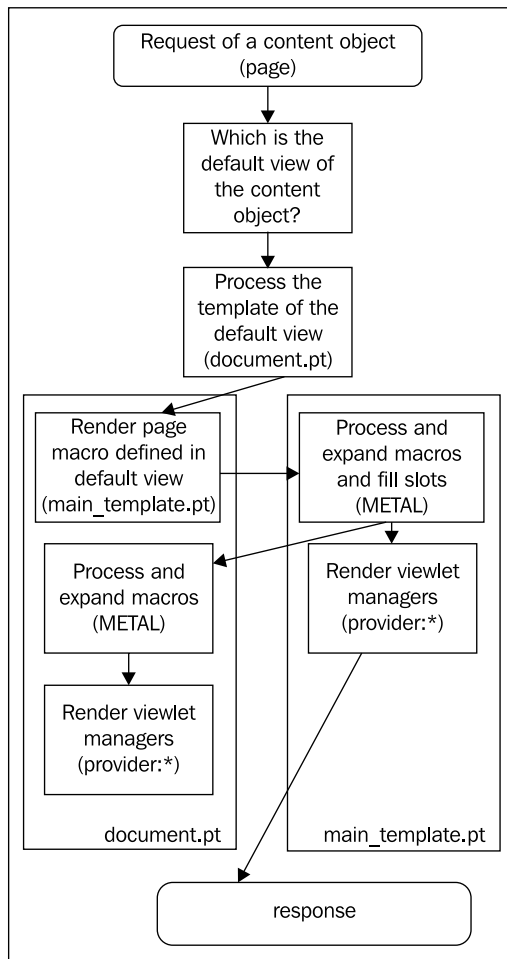


Diving into Plone's page rendering

Plone combines the power of two types of resources into its core – the old-school Plone's skin layers and Zope3-style resources, such as views and viewlets. Plone uses them to add functionality, presentation, and theming (usually known as skinning) to its sites. Let's see what happens when Plone renders a content object. We will come across key theming concepts where we will make a break to explain these properly.

Let's assume a browser requests a URL from our site. Let's say this URL points out to a content object, or a page to be more precise. Consider we have a page in a folder, for example, `http://localhost:8080/intranet/a-folder/a-page`.

Following is a diagram of the workflow when Plone renders the page:



Each time Plone receives a request for rendering a content object, it determines the default view for that object. This value is stored as an attribute in the object and is set (if several views for a content type are defined, for example in a folder) in the display drop-down menu. The default view is in fact a Zope page template object, represented by a `.pt` file. This file is usually stored in a Plone skin layer and, in the case of page content type, it's called `document_view.pt`. It's located in the `plone_content` skin layer.

Plone processes the `document_view.pt` template and determines that it contains a whole page macro that should be processed as well. This macro is in `main_template.pt`, located in `plone_templates` skin layer. Plone composes the resultant page by processing both templates in parallel. Both templates contain references to macros and viewlets that should be expanded and rendered. They will provide the pieces of code that will compose our resultant Plone page.

Before we can continue with the workflow, we need to know more about Plone skin layers, Zope page templates, viewlets, and a fundamental key concept in Zope – acquisition.

Acquisition from parents

This is a key concept in Zope and it's a part of its very core. Any Zope object can acquire any object or property from any of its parents. Let's say we have a folder called **a-folder** that contains another folder called **a-subfolder**. A document called **a-page** exists in our intranet site root:



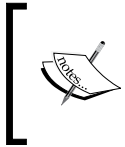
A URL pointing to `http://localhost:8080/intranet/a-folder/a-subfolder/a-page` would work even if a subfolder was empty. This happens because Zope starts to look for the page where we pointed it out in the first place. If it doesn't find the page, it goes back up to its parent where it's finally found and is returned correctly.



The complete explanation about acquisition is far more complex. We can find out more about acquisition in the Zope documentation at this URL: <http://docs.zope.org/zope2/zope2book/Acquisition.html>.

Plone skins tool

This tool, located at the root level of our Plone site, controls most of Plone's skin engine, if not all. It extends the user interface of Zope's **Content Management Framework (CMF)** by providing configurable and layered name lookup to Zope resources, which may be any element involved with theming (templates, images, scripts, and so on) made available to the content on the site in a controlled fashion.



CMF is an add-on product for Zope to build Content Management Systems. It provides some basic tools for handling metadata, members, and so on, but is not a CMS itself. Plone is an example of a sophisticated CMS built using the CMF.

Skins and layers

Let's take a look at the Plone skins tool. Open our site ZMI and access the `portal_skins` tool. The default tab is **Contents**. In this view, we should see all the layers available to the site. They appear to us as Zope folders although they, in fact, represent real filesystem folders from Zope and Plone products. As mentioned before, they contain Zope2-style resources such as page templates, CSS files, images, Python scripts, and other support resources. These resources are needed either to add functionality to a Zope/Plone add-on product (and of course, Plone itself) or they are directly involved in the theming of our site.

Type Name	Size	Last Modified
<input type="checkbox"/> ATContentTypes		2010-03-16 22:45
<input type="checkbox"/> ATReferenceBrowserWidget		2010-03-16 22:45
<input type="checkbox"/> CMFEditions		2010-03-16 22:45
<input type="checkbox"/> ChangeSet		2010-03-16 22:45
<input type="checkbox"/> LanguageTool		2010-03-16 22:45
<input type="checkbox"/> PasswordReset		2010-03-16 22:45
<input type="checkbox"/> ResourceRegistries		2010-03-16 22:45
<input type="checkbox"/> archetypes		2010-03-16 22:45
<input type="checkbox"/> archetypes_kss		2010-03-16 22:45
<input type="checkbox"/> cmf_legacy		2010-03-16 22:45
<input type="checkbox"/> cmfeditions_views		2010-03-16 22:45
<input type="checkbox"/> custom		2010-02-07 19:13
<input type="checkbox"/> gruf		2010-03-16 22:45
<input type="checkbox"/> gruf_plone_2_0		2010-03-16 22:45
<input type="checkbox"/> kupu		2010-03-16 22:45
<input type="checkbox"/> kupu_plone		2010-03-16 22:45
<input type="checkbox"/> kupu_references		2010-03-16 22:45
<input type="checkbox"/> kupu_tests		2010-03-16 22:45
<input type="checkbox"/> mimetypes_icons		2010-03-16 22:45
<input type="checkbox"/> plone_3rdParty		2010-03-16 22:45
<input type="checkbox"/> plone_content		2010-03-16 22:45
<input type="checkbox"/> plone_deprecated		2010-03-16 22:45
<input type="checkbox"/> plone_ecmascript		2010-03-16 22:45
<input type="checkbox"/> plone_form_scripts		2010-03-16 22:45
<input type="checkbox"/> plone_forms		2010-03-16 22:45
<input type="checkbox"/> plone_images		2010-03-16 22:45
<input type="checkbox"/> plone_kss		2010-03-16 22:45

Plone tends to organize its layers by categorization (**plone_content**), by the types of elements they contain (**plone_images**), or grouped by add-on products (**CMFEditions**). They may be a part of the Plone core (as in these examples) or not. Third party add-on products usually add layers in order to contain the resources they may need.

A skin is a named group of layers in a specific order of precedence. We can see the definition of all the available skins in the site in the **Properties** tab:

The screenshot shows the 'Plone Skins Tool' interface at the URL `/intranet/portal_skins`. The interface has a navigation bar with tabs: Contents, Properties, View, Security, Undo, Ownership, Interfaces, Find, Overview, and Actions. The main content area is titled 'Skin selections' and contains a table with two columns: 'Name' and 'Layers (in order of precedence)'. The 'Plone Default' skin is selected, and its layers are listed in a scrollable box: custom, LanguageTool, cmfeditions_views, CMFEditions, ChangeSet, kupu_plone, kupu, kupu_tests, and archetypes. Below the table are 'Delete' and 'Save' buttons. The 'Add a new skin' section has input fields for 'Name' and 'Layers', and an 'Add' button. The 'Default skin' is set to 'Plone Default'. The 'REQUEST variable name' is 'plone_skin'. There are two checkboxes: 'Allow arbitrary skins to be selected' (unchecked) and 'Make skin cookie persist indefinitely' (unchecked). A 'Save' button is at the bottom.

Name	Layers (in order of precedence)
<input type="checkbox"/> Plone Default	custom LanguageTool cmfeditions_views CMFEditions ChangeSet kupu_plone kupu kupu_tests archetypes

Add a new skin

Name	Layers
<input type="text"/>	<input type="text"/>

Default skin Plone Default

REQUEST variable name plone_skin

Skin flexibility Allow arbitrary skins to be selected

Skin Cookie persistence Make skin cookie persist indefinitely

Obviously, Plone ships with a default skin called **Plone Default**. The most interesting part of this view is the upper part. Here, we can see all the skins installed in the site and the layers associated to each of them. The higher the layer in the list, the higher the precedence of that layer. We could change the precedence or make a specific layer available or unavailable by modifying this list.

Acquisition in skin layers

The resources contained in the active layers of the current skin are available on our site from our browser regardless of the level of the folder we may be accessing in the hierarchy. It's like they were located in the root site folder. Let's assume we have a CSS file in one layer and it's called `style.css`. It will be available from any folder in our site, thanks to the magic of acquisition. Back to the example used in the acquisition section, `style.css` will be available from:

- `http://localhost:8080/intranet/a-folder/a-subfolder/style.css`
- `http://localhost:8080/intranet/a-folder/style.css`
- `http://localhost:8080/intranet/style.css`

Zope page templates

Zope page templates are objects that allow us to define dynamic presentation for a website. The HTML in our template is made dynamic by inserting special XML namespace elements into our HTML code, which define the dynamic behavior for that page. Zope executes ZPT commands on the server, and the results are sent to our web browser in plain HTML.

A common problem arises with many languages designed for creating dynamic HTML content – they don't allow proper separation of presentation and logic. For example, some scripting languages, such as JSP, PHP, or SSI, embed special tags into HTML that can confuse or disorient graphic designers who don't know much about creating an application around the design they are generating. These tags can sometimes cause the HTML to become invalid for the design tools they commonly use.

TAL

ZPT uses an attribute-based presentation language that tries to allow round-tripping of templates between programmers and non-technical designers. This language is called TAL.

TAL consists of a set of special tag attributes, whose name starts with `tal:`, and the values associated with them. The values of a TAL statement are shown inside quotes. For example, this code snippet will render the title attribute of the context using the `tal:content` tag:

```
<p>
  The title of the current content object is
  <b tal:content="context/Title">the title</b>
</p>
```

If the title of the context is `A page`, this snippet will render:

```
The title of the current content object is A page.
```

These are some of the available TAL tags:

- `tal:content` and `tal:replace`: For inserting text and structure
- `tal:define`: For defining variables inside the templates
- `tal:repeat`: For looping through data structures, such as lists
- `tal:condition`: Used to control the display of a block
- `tal:attributes`: For dynamically setting tag attributes

METAL

ZPT also allows defining macros with tags; attributes similar to TAL statements. They are called **Macro Expansion Tag Attribute Language (METAL)**. Macros provide a way to define a chunk of presentation structure in one template, and share it in another. It can be a section of a page or an entire page.

Macros lets us define some parts that can be overridden when we use them. We can do this by defining slots in the macro that we can fill in later from another template.



Insights about TAL, METAL, and ZPT are out of the scope of this chapter. They are huge subjects; here we will cover only the simplest aspects and how to get started with these through some examples in the next sections. However, you can find a complete manual about ZPT online in the Zope book documentation at <http://docs.zope.org/zope2/zope2book/ZPT.html>, and a complete reference manual for TAL expressions at <http://docs.zope.org/zope2/zope2book/AppendixC.html>.

Viewlets

Viewlets are grouped in **Viewlet Managers**, and use a Zope3 concept called content provider, which renders a collection of viewlets that are registered for it. Content providers are small pieces of HTML code generated by an auxiliary template (in this case, a Zope3 view). They can be called from a ZPT template or from another view. This substitutes the old Zope2 approach of using macros to include in `main_template` pieces of HTML generated by other templates. Following is the list of the most interesting viewlets:

1. `plone.siteactions`
2. `plone.searchbox`

3. `plone.logo`
4. `plone.global_sections`
5. `plone.personal_bar`
6. `plone.path_bar`
7. `plone.contentviews`
8. `plone.contentactions`
9. `plone.belowcontenttitle.documentbyline`
10. `plone.abovecontenttitle.documentactions`
11. `plone.footer`
12. `plone.colophon`

Their locations in a Plone site are shown in the following screenshot:

The screenshot shows a Plone site interface with the following elements and their locations marked with red numbers:

- 1**: Site Map, Accessibility, Contact, Site Setup (top right navigation)
- 2**: Search Site (search bar)
- 3**: Plone logo (top left)
- 4**: Home, Users, News, Events (top navigation bar)
- 5**: admin, Log out (user profile)
- 6**: You are here: Home (breadcrumb)
- 7**: Contents, View, Edit, Rules, Sharing (action buttons)
- 8**: Display, Add new..., State: Published (action buttons)
- 9**: by admin — last modified May 23, 2010 07:04 PM (author and date)
- 10**: Send this — Print this — (bottom right of content area)
- 11**: The Plone CMS — Open Source Content Management System is © 2000–2010 by the Plone Foundation et al. (copyright notice)
- 12**: Powered by Plone, Valid XHTML, Valid CSS, Section 508, WCAG (bottom footer)

The main content area displays a welcome message and instructions for getting started. A calendar for May 2010 is visible on the right, with the date 23 highlighted. A "History" portlet is also present at the bottom of the content area.

Managing viewlets

Plone organizes viewlets in viewlets managers. Viewlets in Plone can live only inside viewlet managers. This approach gives us the advantage to organize different regions of a Plone page without having to modify the code in `main_template.pt`.

Plone has a managing facility that allows site managers to perform some actions over viewlets. We can access this special view for managing viewlets by accessing the following URL: `http://localhost:8080/intranet/@manage-viewlets`.

We can perform some interesting actions over viewlets from this view. We can move the position of any viewlet up or down, though only within the limits of its own viewlet manager. We can do so by using the arrow links beside the viewlet name.

We can also choose to hide or show any viewlet using the **hide/show** link beside the position arrows. The following screenshot shows how it looks:



Composing a Plone page

Now, we have enough background to return to the initial example and learn how Plone renders a page. Following the workflow diagram shown at the beginning of this section, once Plone knows the default view of a content object, it processes it. In the example, it processes the `document_view.pt`, located in the `plone_content` skin layer. Search for it in the `portal_skins` tool inside the `plone_content` layer.

If we open it, we will see that it begins like this:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      lang="en"
      metal:use-macro="here/main_template/macros/master"
      i18n:domain="plone">
```

At the beginning of the template, we declare all the namespaces needed in the template. After doing this, the other external tools we may use to edit this resource (such as developer or designer tools) will not complain about unrecognized XML tags. We do all this in the `html` tag.

The most important attribute in this piece of code is the `metal` statement. It tells Plone that this template uses a whole page macro. This macro is located in the `main_template.pt` macro called `master`. As `main_template.pt` is in the acquisition chain, `document_view.pt` can refer to it as `here/main_template`.

The rendering of the page continues processing the `master` macro in the `main_template.pt` page template.

Rendering the `main_template.pt` page template

This template defines the whole basic structure of any Plone page. It will:

- Render all the needed headers
- Pull the CSS and JavaScript needed to render the page
- Call several viewlet managers to make them render the viewlets assigned to them
- Render the existing portlets for the current context
- Return the render process to the template that has called it previously (`document_view.pt` in our example) to render the `main` slot
- Close the page with the footers

It will use ZPT macros and viewlets to do all these tasks.

The most important slot defined by `main_template.pt` is `main` slot. This slot is reserved for including the information of the object being rendered. This slot is filled by information defined by the template that had previously called `main_template.pt`, depending on the content type or view we are requesting to Plone. For example, if we are requesting a page content type, `main_template.pt` will render all the common elements to that object and then call the default view of the page content type to render the contents of that page.

We will find it in the `plone_templates` skin layer; let's open it and have a look. Following is a snippet of its contents:

```
<metal:bodytext metal:define-slot="main" tal:content="nothing">
  Page body text
</metal:bodytext>
```

This will define the `main` slot in `main_template.pt`. Going back to `document_view.pt`, this is the code corresponding to this macro:

```
<metal:main fill-slot="main">
  <tal:main-macro metal:define-macro="main"
    tal:define="kssClassesView context/@@kss_field_decorator_view;
    getKssClasses nocall:kssClassesView/getKssClassesInlineEditable;
    templateId template/getId;
    text here/getText;">
```

This will define the `fill` slot and other attributes and variables needed for the template to work. This will call the `plone.abovecontenttitle` viewlet manager:

```
<div tal:replace="structure provider:plone.abovecontenttitle" />
```

This other piece of the template calls the title of the content being viewed. It uses an `<h1>` tag and a macro to retrieve the correct structure for this content field:

```
<h1 class="documentFirstHeading">
  <metal:field use-macro="python:here.widget('title', mode='view')">
    Title
  </metal:field>
</h1>
```

After some other content field definitions, we reach the end of the template:

```
[...]
<div metal:use-macro=
  "here/document_relateditems/macros/relatedItems">
  show related items if they exist
```

```
</div>
  <div tal:replace="structure provider:plone.belowcontentbody" />
</tal:main-macro>
</metal:main>
```

Again, it calls a macro to render the related items of the content object and then calls the viewlet manager `plone.belowcontentbody`.

Now, we know how Plone renders a page content type. Plone deals with other kinds of resources, such as templates not bound to content, (for example, `sitemap.pt`) but they are generated in pretty much the same manner as content-bound templates.

There are other types of resources such as CSS and JavaScript files that require special treatment in Plone. They are managed via **Resource registries**.

Resource registries

Plone provides a facility called resource registry. It's used in different contexts to effectively manage different kinds of resources, such as CSS and JavaScript files. They provide caching, conditional enable/disable, merging, and maintain a list of order of precedence for the resources. It is usual to find these resources as Plone skin layer resources in `portal_skins`, but it's also possible to find these as Zope3-style resources too, living in the Zope3 machinery.

When Plone renders any object, it calls the resource registries to retrieve the CSS and JavaScript registered for the site. Then, resource registries will, if applicable, merge and cache them.

CSS resource registry

These are located in the `portal_css` tool in the ZMI site root. Here, we can find all the CSS registered in our site. The inclusion of these resources in the final page rendered is being determined by the conditional expressions, if any. This tool provides merging of CSS files and caching capabilities, as well as an improved page rendering performance.

This tool also provides order for the inclusion of these CSS files in the resultant HTML. Remember that in the case of CSS files, they are applied in cascade, and the last declared one takes precedence over the first ones. So styles declared in the last positions of this list of CSS files will take more precedence over other styles.

JavaScript resource registry

This registry is located in the `portal_javascript` tool in the ZMI site root. Here we will find all the JavaScript files registered for the site. The behavior of this registry is similar to the CSS resource registry.

Theming using third-party add-on products

Now we have all the background needed to begin customizing our intranet. We know how Plone renders any object, the elements involved in the rendering, and the different kind of resources Plone uses.

We will cover several approaches: using third-party applications and using a custom theme add-on product.

GloWorm—add-on product for viewlet customization

Customizing viewlets is a complex thing and it's mainly done by programming. However, there are some tools that ease the process. Of course, there are some advanced things we can't do on the web and need more advanced knowledge. However, basic customizing covers a lot of use cases, so let's get started.

There is a splendid tool for theming our site and making changes to viewlets and other elements called `Products.Gloworm`. Written by Eric Steele, the release manager of Plone 4, it's aimed to ease the development of Plone sites.

Installation

We can install it by including these lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
...
eggs = Products.Gloworm
```

Then, rerun `buildout.cfg`:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Now, go to the **Add-on Products** control panel configlet and install it.

Using GloWorm

GloWorm requires running our Zope instance in debug mode to work correctly. It's enough if we've started our instance in foreground (`fg`) mode. GloWorm is intended to be a development tool and is best left uninstalled on our production machine.

An **Inspect this page** link will appear in the **Object Actions** section of content objects on our site. In a typical Plone installation, this is at the bottom of the page content, besides the **Print this** link. We may also access this view by appending `@@inspect` to the current page's URL.

If we click on this link, then the **GloWorm Inspector Panel** is shown at the bottom of the page. In the inspector view, clicking on any element of the current page will bring up a list of information about that page element, including TAL statements and the name of the viewlet or portlet in which it is contained:

The screenshot shows a web browser window displaying a Plone site. The address bar shows the URL `http://localhost:8080/intranet/front-page/@@inspect`. The page content includes a 'Welcome to Plone' message and a 'Get started' section. The GloWorm Inspector Panel is open at the bottom, showing a tree view of the page structure. The selected viewlet is `plone.path_bar`, and the details panel shows the following information:

Property	Value
Name	<code>plone.path_bar</code>
Manager	<code>plone.portaltop</code>
Class	<code>plone.app.layout.viewlets.common.PathBarViewlet</code>
Template	<code>zope.interface.interface-plone.path_bar</code>
Visibility	Visible <input type="checkbox"/> Hide viewlet <input type="checkbox"/>

There is also a 'Customize' button at the bottom of the details panel.

In the previous screenshot, the **plone.path_bar** viewlet is selected. We can see information about its manager (**plone.portaltop**) and the current visibility of this viewlet. We can customize it from the inspector by clicking on the **Customize** button.

GloWorm's template customization feature utilizes Plone's **portal_view_customizations** utility to manage viewlet templates. To find our customized templates in the ZMI, go to the **portal_view_customizations** tool within our Plone site and select the **Contents** tab.

Inspecting the viewlet managers will also show us an interface for reordering viewlets contained by this manager and **show/hide** controls as well. Clicking the up and down arrows performs viewlet reordering. Clicking the name of a viewlet will take us to the viewlet inspection view for that viewlet.

Click the red close icon in the upper-right corner of the GloWorm panel to browse the site normally.

CSSManager—add-on product for CSS and basic properties customization

This product provides a UI for changing some basic properties of our site, such as colors, borders, and other visual attributes. No other special knowledge of CSS is needed.

Installation

We can install it by including these lines in the specified sections of our `buildout.cfg` file:

```
[buildout]
..
eggs = Products.CSSManager
```

And then, rerun `buildout.cfg`:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Now, go to the **Add-on Products** control panel configlet and install it.

Using CSSManager

CSSManager installs an additional configlet in the Plone control panel. We should turn on customization before we make any change. Just click on the **Turn on customization** button.

We should also enable **CSS debug** mode to see changes immediately. We can change this setting using the **CSSManager** configlet, since it exposes a UI for this feature located in the CSS resource registry ZMI. For more information, see the *Enabling CSS debug mode* section later in the chapter.

However, remember to disable it when we finish our customization; otherwise, the site performance will be slower. The product configlet looks like this:

Theme Configuration Manager

[^ Up to Plone Setup](#)

CSS Manager lets you change the colors, borders, site logo and other key visual characteristics of your site.

Enable / disable CSS Debug Mode

CSS Debug Mode is currently **ENABLED**: your changes will take effect right away, but your site runs a little more slowly than usual.


Disable CSS debug mode to make your site run at its usual speed.

Changing the look of a site that's already live? We recommend you disable CSS Debug Mode as soon as you finish making changes here.

Things you can customize through this interface

Technical note for site administrators: changing things here writes to the property sheet in the `portal_skins/custom` folder.

Theme name
The theme you are customizing: **Plone Default**

Site logo
Upload a file from your computer to use as the logo for the site. 

Main font
The preferred font, and the fall-backs if the preferred font is not available in the user's browser.

Default text size
You can specify this as a % value, or in em, or (not recommended) in pixels.

Default text color
The color for text in the content well of the page. Colors can be specified as words or hex values.

Small text size
If you specify this as a % value, it is a % of the default text size (defined above).

These are some of the basic theming properties that can be managed with CSSManager:

- Default text color
- Default text size
- Main font
- Page background
- Default border color
- Default border width
- Logged-in tabs border color

CSS customization with base_properties sheet

CSSManager takes advantage of the fact that Plone exposes its CSS resources as **Document Template Markup Language (DTML)** files. DTML is a dynamic template language and is the precursor of ZPT. By using these kind of files to render out-of-the-box CSS files, Plone made dynamic CSS possible on its sites. This has the advantage of allowing us to modify the basic aspects of our site without the modification of any resources. This is done using a special properties sheet called **base_properties** located in the **portal_skins** tool, inside the **plone_styles** layer.

As CSS are DTML files, these properties are hardcoded as DTML variables. DTML will look up these variables in `base_properties` and generate the resultant CSS file with the `base_properties` values included.

CSSManager exposes a UI only for the customization of this resource and other attributes like the CSS debug mode. We could customize this resource by hand, but it's more easy to use CSSManager to manage it.

Changing the logged-in tabs' attributes

We can easily change the logged-in tabs' look and feel by modifying three attributes in CSSManager. Just modify the values of these attributes:

- Logged-in tabs border color: #999999
- Logged-in tabs background color: #EDED
- Logged-in tabs text color: #999999

Apply the changes and we will get rid of that ugly green color.



Unfortunately, the use of DTML has been deprecated and Plone 4 doesn't use it anymore to define CSS files. For this reason, CSSManager will work only with Plone 3.

Custom theme add-on products

There are lots of custom Plone theme add-on products listed in plone.org. We can always choose one of them to theme our site. Yet, if there are things we want to customize, it's always a good idea to begin with one of them.

A theme add-on product is installed as any other third-party product. Just search on <http://plone.org/products> for a suitable one. We can narrow our search by selecting **Themes** in the category drop-down box.

Building our own theme add-on product

We can build our own theme add-on product from scratch. It can be done easily with PasteScript. Just type this on our command line from the root of our buildout:

```
$ cd src
$ paster create -t plone3_theme plonetheme.myintranet
```

Answer all the questions with the default option except for:

- **Skin Name:** A human facing name for the theme, added to `portal_skins`, for example, `MyIntranetTheme`.
- **Skin Base:** Name of the theme from which this is copied. By default, it is **Plone Default**. Answer the default option here.
- **Empty Styles?:** If true, it will override default public stylesheets with empty ones. Answer **False**.
- **Include documentation?:** If true, the generated theme will include auto-explanatory documentation, desirable for beginners.

The resultant theme add-on product will be generated in the `src` buildout folder. This add-on is completely usable right now, but it's innocuous. Once installed, it will replace the original Plone default theme with the one in this package.

Installing the product

Just proceed as any other add-on product. However, since we are developing the product, we should specify it in our buildout by filling the `develop` directive in the `buildout` section and the `eggs` directive in the `instance` section in our `buildout.cfg` file:

```
[buildout]
...
develop = src/plonetheme.myintranet
...
[instance]
...
eggs = plonetheme.myintranet
```

Go to the package folder, `src/plonetheme.myintranet/plonetheme/myintranet`, and edit the `configure.zcml` file. As we don't want to define an `i18n` folder, delete the following line if it exists:

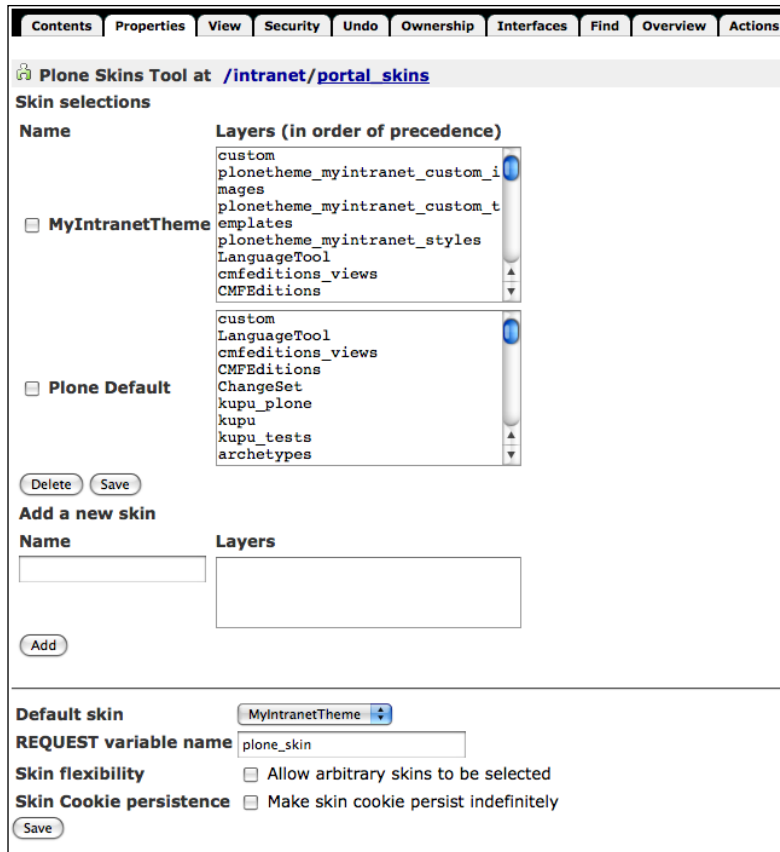
```
<i18n:registerTranslations directory="locales" />
```

And then, rerun `buildout.cfg`:

```
$ ./bin/buildout
$ ./bin/instance fg
```

Now, go to the **Add-on Products** control panel configlet and install it.

If we browse our site, we will notice that nothing has changed, because we've chosen to inherit the default theme in our new one. But, now the theme defined in our theme add-on product is in use in our site. Check it out in **portal_skins**:



Notice three things in the previous screenshot: the **Default skin** is our recently created skin and three additional Plone skin layers have been added to the top of the layer's precedence order list. These three layers will contain the resources we may need for our new theme. These layers represent three folders inside our package structure; to be more precise, those inside `skins` folder:

Name of the layer/folder	Description
<code>plonetheme_myintranet_custom_images</code>	It will contain our theme images.
<code>plonetheme_myintranet_custom_templates</code>	It will contain our theme custom templates.
<code>plonetheme_myintranet_styles</code>	It will contain our theme styles.

In fact, this layer organization is merely for convenience, as all the layers can contain any type of resources.

Customizing Plone skin layer resources

As our theme product is positioning the new layers on the top of the precedence order, the elements we place in these folders will override those in layers with less precedence. Just place our custom resource in any of the layers defined by our product and name it as the original one. Our custom resource will override the default one. We can also place other resources we may use, such as our custom templates, images, and styles as well.

Enabling CSS debug mode

By default, the changes made to our product will not be available until we restart our instance. For the changes to take effect immediately, we should enable CSS debug mode in **CSS resource registry**. We will find this setting at the top of the `portal_css` ZMI view.

In debug/development mode, stylesheets are not merged to composites, and caching and compression of CSS is disabled. The registry also sends HTTP headers to prevent browsers from caching the stylesheets. It's recommended to enable this mode during CSS-related development. Remember to turn it off again when we finish CSS modifications, as debug mode affects site performance.

Customizing the site logo

Plone renders the site logo combining two kinds of resources – the viewlet `plone.logo` provides the HTML structure needed and a Plone skin layer image. Let's say we want to change the site logo and add an additional logo of our company containing a link to the corporate web besides it. We need to customize the original logo with the logo of our intranet and add the required HTML structure to add the new company logo besides the original one. We will need to customize the original logo and the `plone.logo` viewlet. Later, we will need to add our company logo as a new Plone skin layer image.

Customizing the logo image and adding a new one

We should override the original logo image with our customized one. In order to accomplish this, we should rename the image we've chosen to use as our site logo with the same name as the original one. The original logo image is called `logo.jpg` and it is located in the `plone_images` skin layer. We override it by simply placing our customized image inside `skins/plonetheme_myintranet_custom_images` and naming it exactly the same as the original one. Place the image for the second logo here too, and name it as `company-logo.png`.

Customizing the plone.logo viewlet

Customizing a viewlet is a little trickier than overriding skin layer resources. We will need to tell Zope that we want to override the original viewlet declaration by creating an `overrides.zcml` file in the `plonetheme/myintranet` folder of our custom add-on product, and add the attribute that tells Zope where to find the new template associated to this viewlet:

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  xmlns:i18n="http://namespaces.zope.org/i18n"
  i18n_domain="plonetheme.myintranet">
  <!-- The new logo viewlet declaration -->
  <browser:viewlet
    name="plone.logo"
    manager="plone.app.layout.viewlets.interfaces.IPortalHeader"
    class="plone.app.layout.viewlets.common.LogoViewlet"
    template="browser/newlogo.pt"
    permission="zope2.View" />
</configure>
```

Then place this Zope page template called `newlogo.pt` in the browser folder of our add-on product:

```
<a id="portal-logo-company"
  tal:attributes="href string:http://www.mycompany.com/">
  </a>
<a metal:define-macro="portal_logo"
  id="portal-logo"
  accesskey="1"
  tal:attributes="href view/navigation_root_url"
  i18n:domain="plone">
  </a>
```

We leave the original logo template at the end of the file and add a new link tag with the structure for the new logo and the reference to the new Plone skin layer image (`company-logo.png`).

Restart our instance to see the changes applied. This is needed because we have overridden a viewlet defining an additional ZCML file.

Customizing Plone CSS

If you followed the previous example, you've probably noticed that the recently added new logo is somehow displaced. If we want to adjust its position, we will need to customize some CSS files. In this case, we need to define a style for the new ID (`portal-logo-company`) introduced by the structure of the company logo. We can do it in an empty CSS file provided by the `PasteScript` template, already available in our theme product. It's called `main.css` and it's located in the `plonetheme/myintranet/browser/stylesheets/` folder.

Open it and add these lines:

```
#portal-logo-company img {
    border:0 none;
    margin:1em 0 1em 2em;
    padding:0;
}
```

We can also adjust some other CSS declarations by adding them to this file. As `main.css` is located at the bottom of the CSS chain, the definitions in it will be the last being applied. For this reason, we can override any previous CSS definitions by redefining them in this file.

For example, let's adjust the space between both logos by reducing the margin defined by the `portal-logo` ID. Add these lines to `main.css` file:

```
#portal-logo img {
    margin:1em 0 1em 1em;
}
```

If you want to know more about Plone CSS and the resources involved in Plone styling, you should check the CSS files located in the `plone_styles` layer. They are organized by importance, styled elements, or purpose. The following table gives a little summary about them:

Name of the CSS resource	Description
<code>member.css.dtml</code>	Plone member specific CSS, such as styling for the colors assigned to workflow states in the state drop-down box.
<code>base.css.dtml</code>	Plone basic elements CSS, such as styling for basic tag elements (<code>a</code> , <code>body</code> , and so on)
<code>public.css.dtml</code>	Plone public facing elements CSS, such as style for the content headers (<code>h1</code> , <code>h2</code> , and so on).
<code>columns.css.dtml</code>	Plone table-based layout CSS.
<code>authoring.css.dtml</code>	Plone authoring/editing environment CSS.

Name of the CSS resource	Description
<code>portlets.css.shtml</code>	Plone portlets CSS.
<code>controlpanel.css.shtml</code>	Plone control panel CSS.
<code>print.css.shtml</code>	Plone print CSS applied when printing some site element.
<code>deprecated.css.shtml</code>	Deprecated Plone CSS elements that will disappear in the next version.
<code>navtree.css.shtml</code>	Plone navigation tree CSS.
<code>invisibles.css.shtml</code>	Plone invisible elements and accessibility elements CSS.
<code>forms.css.shtml</code>	Plone forms CSS.
<code>RTL.css.shtml</code>	Plone right-to-left CSS (for Arabic and Hebrew).
<code>ploneCustom.css.shtml</code>	Blank CSS file ready for quick style customization.



Since Plone 4 no longer uses DTML files to define CSS, if we want to customize them, we should just remove the `.shtml` extension from the previous files.

Resetting Plone CSS

If we want to reset the CSS defined by Plone completely, we should include these empty files in our product skin layer folder `plonetheme/myintranet/skins/plonetheme_myintranet_styles`:

- `base.css.shtml`
- `public.css.shtml`
- `portlets.css.shtml`

By doing this, we are overriding the original files with empty files, so the defined styles by these CSS are not applied. Once reset, we can freely redefine all the Plone styles without inheriting any of them.



How to customize CSS effectively

It's recommended to use some kind of helper application when dealing with CSS, for example, Firebug (<http://getfirebug.com/>). This tool will help us to identify the correct CSS file used by any element of the site and preview the results of any change to the CSS on the fly. Don't forget to enable debug/development mode in your `portal_css` resource registry!

We will find this product along with the code in this example in the code bundle for this chapter.

More about customizing viewlets

All the default viewlets are defined in a Plone egg called `plone.app.layout`. We can find it by browsing the `eggs` folder inside our `buildout.cfg` folder. We can override any of them following these steps:

1. Use the `manage-viewlets` view to identify the viewlet we want to customize.
2. Search the viewlet declaration in the `configure.zcml` file inside the `plone.app.layout-1.2.5-py2.4.egg/plone/app/layout/viewlets` folder. For example, if we want to customize the portal breadcrumbs located in the `plone.path_bar` viewlet, find the snippet of code referring to this viewlet. It should look like this:

```
<!-- The breadcrumbs -->
<browser:viewlet
    name="plone.path_bar"
    manager=".interfaces.IPortalTop"
    class=".common.PathBarViewlet"
    permission="zope2.View" />
```

3. Copy these lines to the `overrides.zcml` file and modify them to look like:

```
<!-- The new site breadcrumbs declaration -->
<browser:viewlet
    name="plone.path_bar"
    manager="plone.app.layout.viewlets.interfaces.IPortalTop"
    class="plone.app.layout.viewlets.common.PathBarViewlet"
    template="browser/newpathbar.pt"
    permission="zope2.View" />
```

4. As we are overriding a viewlet located in another egg product, we should complete the prefix in the `manager` attribute with the full path to the `plone.app.layout` egg. Do the same for the `class` attribute.
5. Add a line with a `template` attribute defining where the new template for this viewlet is located. Its location is relative to our product folder. In the example, we are defining a new template for the viewlet called `newpathbar.pt` located in the `browser` folder. It's recommended that we copy the default template defined by `plone.app.layout` and customize it. We will find it in the `plone.app.layout-1.2.5-py2.4.egg/plone/app/layout/viewlets` folder; it's called `path_bar.pt`.

Using Generic Setup to customize a theme

Once more, Generic Setup (GS) has some configuration files that help us to customize some theming aspects of our site. We can add the following GS definition files to the default profile of our product:

Generic Setup file	Description
<code>cssregistry.xml</code>	It configures the CSS resource registry (<code>portal_css</code>). We can add more CSS resources and configure them. By default, Paster defines the <code>main.css</code> file and places it at the bottom of the CSS chain.
<code>jsregistry.xml</code>	It defines the JavaScript resource registry (<code>portal_javascript</code>).
<code>viewlets.xml</code>	It configures the position and the visibility of the site viewlets. Basically we can perform all the tasks provided by <code>manage-viewlets</code> view.
<code>skins.xml</code>	This file holds the setup configuration for the <code>portal_skins</code> tool. We can enable new layers and position and order them in the layer precedence list.

Theming—best practices

Whilst theming for an intranet, it is important to keep one thing in mind: try to keep it as clean and simple as possible for one thing only — performance. Most of the time, our intranet will be used by authenticated members. This means that all the security calculations, template generating, and the JavaScript needed in the edit mode will be at work.

Try to minimize their effects by not bloating our intranet theme. It will be always well-received by our users and their subjective sense about how our site performs. Think that an intranet usually doesn't have to sell anything, as a public site would do. Probably the only thing we have to sell to our intranet users is value, productivity, elegance, efficiency, and a corporate look and feel. So, try to deliver all of these to them!

This doesn't mean that our intranet has to be austere, ugly, or boring. On the contrary, it's just a matter of finding the correct balance and applying it.

Summary

This chapter has covered:

- How Plone performs theming and the technologies involved in the process
- The best practices on theming an intranet
- Useful examples and recipes to give immediate value

The next chapter will cover how to deploy our intranet in our production servers.



If you want to learn more about Plone theming, there is an outstanding Packt Publishing book "*Plone 3 Theming*" (2009), *Veda Williams*, covering it in detail.

13

Deploying our Intranet

Time for the final stage for our intranet. We've built it, chosen its functionality, and designed its structure and security model. After we have reached this point, the only thing left to do before our intranet goes live is to execute the deployment of our site onto our production server.

Deployment must be thoughtful and well executed. It is not an easy step, on the contrary, this is probably the most important part of the intranet build process and the one which would require more care from us. It's not a single action; it's more of a continuous process. We should monitor our site in case we have to change our type of deployment, because our site requirements may change through its life cycle.

The keys for a successful deployment are:

- Potential user access estimation
- Potential peak concurrent users estimation
- Potential data sizing estimation
- Continuous monitoring
- Foresee potential capacity problems and be proactive

Getting the initial estimations right are fundamental to planning the hardware needed and choosing the type of deployment to be used. Whether we are administering the production server or not, it's important that the system administrators, responsible for our service, monitor the service once the intranet is deployed. Knowing in advance that our site load and traffic are increasing and reacting proactively for solving and improving our service is the key to success for our deployment.

Basically, we are going to differentiate between the three types of implementations:

- Small site type: It has a few users (1 to 50), small data stored, and little access load
- Medium site type: It is for mid-sized requirements with 50 to 100 intranet users and medium access load
- Large site type: It is a site with a lot of traffic, high performance requirements, and 100 to N users

In this chapter, we will cover:

- All these types of deployments with examples for each one of them
- Different types of technologies that will be at work to accomplish each type of requirement like `VirtualHostMonster`
- ZEO servers
- Load balancers
- Cache servers and Plone cache
- Using external user databases like LDAP directories

Deployment buildouts

Once again, we are going to delegate our environment building process to buildout. Through this chapter, we will use a default deployment buildout configuration and we will increase its complexity as we introduce you to each type of deployment, along with the different technologies and server applications types.



The buildout structure and configuration type used in this chapter are built following some ideas taken from Martin Aspeli's *Über-buildout*. The approach used in *Über-buildout* is to provide an ordered, clear, multi-platform, and multi-purpose oriented buildout. However, some of its features make it a little bit complex to use and it can be an overkill for some use cases for newcomers – besides that, Martin has done a great job. You are encouraged to take a look at it, as you will be able to extend your knowledge about buildout design. You can find more information about here: <http://www.martinaspeli.net/articles/an-uber-buildout-for-a-production-plone-server>.

Buildout base configuration

All deployment types covered in this chapter will share a common buildout base configuration. This base is very similar to the one that we've been using throughout the book. We will use the configuration generated by PasteScript `create` command as a base template. However, we are going to introduce some changes in order to improve it and make it production ready.

Let's begin by creating a generic buildout:

```
$ paster create -t plone3_buildout deployment.buildout
```

Just answer the default values for all questions except for the admin password. Now, let's improve this buildout configuration.

Adding a versions file

It's always recommended to add a separate file for keeping track and setting the eggs and module versions used in buildout. Buildout will try to download the newest version of each package, egg, and module managed by it, so we must be as explicit as we can in this section and try to include all versions from the eggs we may use. This task is usually referred to as **pin** the egg versions. For example, we should add all the third party add-on product versions we may be using in our intranet. Failing to do so may lead to future version incompatibilities and conflicts.



In times where the transition to Plone 4 is taking place, it's normal to find some eggs and third party modules that implement explicit dependencies on Plone 4. This means that if you add them to your Plone 3 buildout, they will force the installation of Plone 4 (along with its own dependencies, for example Zope 2.12) when your buildout is run. If this happens, your buildout will fail due to versions conflicts and other issues. So take care to pin all your products, eggs, and modules versions to avoid this issue.

We can easily add a versions file (`versions.cfg`) by following these steps:

1. Create a file called `versions.cfg` in the buildout root folder:

```
$ cd deployment.buildout
$ vi versions.cfg
```

2. Add the `versions` section and the following lines to this file:

```
[versions]
# Add here the versions of eggs, modules and add-on products used
in your buildout
# Buildout infrastructure
```

```
plone.recipe.zope2install = 3.2
plone.recipe.zope2instance = 3.6
```

```
# Add-on products
```

```
# modules
```

3. Then add the following line to our `buildout.cfg` file to the `extends` directive:

```
extends =
    http://dist.plone.org/release/3.3.4/versions.cfg
    versions.cfg
versions = versions
```

From now on, our buildout will merge the two `versions` definitions defined in the `extends` directive:

- The one defined by the URL `http://dist.plone.org/release/3.3.4/versions.cfg`
- The one defined in the `versions.cfg` file

The latter will have more precedence than the former one.

Caching extended configuration

Just in case we have to work in the offline mode, we can use a directive to tell buildout to cache the downloaded configuration. Add the following directive:

```
extends-cache = cache
```

The directory structure `cache/extends` should already exist before we run the buildout.

Using the newest directive

There is an interesting directive from the `buildout` section called `newest`. It will help us to force buildout not to download the latest version of an egg. Once our buildout is stable, the directive is recommended to set to `false`. Just add this line to the `buildout.cfg` file:

```
[buildout]
...
newest = false
```

Adding ports and hosts names sections

In order to quickly access host names and ports used by our server processes, it is a good practice to group them in their own sections. Of course, we will have to rewrite some parts of the buildout to accommodate them.

Add these two sections:

```
[hosts]
instance = localhost
```

```
[ports]
instance = 8301
```

Now, modify this line to use these new section settings in the rest of the sections:

```
[instance]
...
http-address = ${hosts:instance}:${ports:instance}
```

Now, we can access and change host names and ports centrally from these new sections without having to search the entire buildout for them.

Adding process owners section

The idea is the same as the previous case. We can add a section to declare the user owners of each server process we will use in our deployment. As a rule of thumb, it's always advisable not to run your server processes as root user. You should always try to use a user without root privileges.

Our policy will be to change the owner of all the server processes that use the same user. To keep it simple, we are going to use the same user to run all processes. We should create this user on our server, or use an existing one of our choice. For example, in the case of using a Linux box, we can do it by using this command:

```
$ adduser plone
```

Answer all the questions with the default option and set the password for this user. Of course, we can use the username we desire. Just make the necessary changes to the following examples accordingly.

Once we've created this new user, we will then add the following new section to our buildout.cfg:

```
[users]
zope = plone
```

And modify the instance section to use this user to launch the Zope process:

```
[instance]
...
effective-user = ${users:zope}
```

Changing the ownership of buildout folder

When we launch our server processes with a non-root user, we should ensure that this user would have full access to the file structure of the buildout. For the vast majority of cases, it's recommended that this user will be the owner of the files and folders of our buildout file structure.

We can force this automatically each time the buildout is run. We are going to use a buildout recipe that runs system commands when buildout is run.

```
[chown]
recipe = collective.recipe.cmd
on_install = true
on_update = true
cmds = echo "Changing buildout owner to plone user"
      chown -R plone.plone ${buildout:directory}
```

And add this line to the parts directive in the buildout section:

```
[buildout]
parts = ...
      chown
      ...
```



We can use this recipe to execute our own OS commands too if we need to do so. Just add our desired commands in the `cmds` directive.

Common administration tasks

We should add some elements to our basic deployment buildout that will help us with the most common system administration tasks. They should be taken into account and implemented by any systems administrator in any deployment.

These tasks have to be scheduled and executed on a regular basis. We can use our operating system task schedule facilities to do so. More on that later in this section.

We have to take care of three necessary tasks: database backup (and restore), log rotation and database pack.

Backing up and restoring database

Backing up our database is probably the most important thing to do whilst doing system administration and back office tasks. Zope provides a single tool for managing backups and restores of the ZODB database. Zope also allows us to perform hot backups of its ZODB databases. This means that we don't have to stop the Zope server to do a ZODB backup. This tool is a Python script called `repozo.py`. Buildout will leave a copy of it in the `bin` folder of our buildout structure.

There is a very convenient buildout recipe for managing backups and restores. It basically installs several wrapper scripts for the `repozo.py` script. We should add these lines at the end of our `buildout.cfg` file:

```
[backup]
recipe = collective.recipe.backup
location = /path/to/your/designated/backupfolder
snapshotlocation = /path/to/your/designated/snapshotbackupfolder
```

And add these lines to the `parts` directive in the buildout section:

```
[buildout]
parts = ...
        backup
        ...
```

Substitute the `location` directive with the folder in which we want to store our instance backups. The recipe will generate these scripts:

Script	Description
<code>./bin/backup</code>	It will perform a backup of our database and will store it in the folder defined by the <code>location</code> directive.
<code>./bin/restore</code>	It will recover the latest backup stored in the backup folder. We can also restore the backup of a certain date. Just pass a date argument. The format is <code>yyyy-mm-dd[-hh[-mm[-ss]]]</code> .
<code>./bin/snapshotbackup</code>	It is for quickly grabbing the current state of a production database in case we want to try something with real data and later download it to our development server or laptop. It will perform a full backup, but without interfering with the regular backup regime. It will be stored in the folder defined by the <code>snapshotlocation</code> directive.
<code>./bin/snapshotrestore</code>	It will recover the latest snapshotbackup stored in the <code>snapshotbackup</code> folder.

Normally, a full backup is done the first time we back up our database. The backup is stored in one single file with the `.fsz` extension (the `z` indicates it's also zipped). Another `.dat` file is also stored with information about the backup and the files involved in it. The next time we perform a backup, an incremental backup will be stored in the same location with the extension `.deltafsz`.

Performing a recovery is an easy operation, but we must provide all the required files for a successful restore. This is the first full backup available previous to the date to which we want to restore the database, and all the incremental files from the day of the full backup to the desired date of recovery.

It's recommended to perform a daily backup, and force a full backup once a week. Forcing a full backup is easy since it's done each time the database is packed.

Database packing

ZODB database is a transaction driven database. This means that every database modification is stored as a transaction and, in theory, can be undone. This is good because we have undo capabilities, but is also bad, because our database will grow with each transaction made. Database packing will consolidate transaction changes into the database, freeing it from all the transactions. This means that we won't be able to undo any transaction, but the size of the current database will shrink considerably depending on the number of transactions stored.

For that reason it is advisable to pack the database from time to time. Doing it once a week is a good practice. Each time the database is packed, a full backup will be triggered when we perform a backup.

Rotating the log files

Another thing that is highly recommended is to truncate log files by days. It's also known as log rotation. Letting the server processes handle small log files can increase server performance. Keeping them small is also good for system administration and debugging problems. On Linux systems, it's something easy to setup through standard **logrotate** daemon. We can generate a `logrotate` configuration file from a buildout template recipe. This is the section we should add to our `buildout.cfg` file.

```
[logrotate.conf]
recipe = collective.recipe.template
input = ${buildout:directory}/templates/logrotate.conf.template
output = ${buildout:directory}/production/logrotate.conf
```

And add this section to the `parts` directive in the `buildout` section:

```
[buildout]
parts = ...
    logrotate.conf
    ...
```

As we can see in the `logrotate.conf` section, we will need a `logrotate` template file. We will need more of this kind of template for configuring other server processes shown in this chapter. For that reason, and to keep it grouped, we will set up a folder for all of them in the `templates` buildout folder. The output file will be stored in a different special buildout folder that we will also create called `production`.

The following is the contents of the `logrotate.conf.template` file:

```
rotate 7
weekly
create
compress
delaycompress
create 644 plone plone

${buildout:directory}/var/log/instance.log {
    postrotate
        ${buildout:directory}/bin/instance logreopen
    endscript
}
```

Buildout will process it, fill the buildout variables in it, and will generate a `logrotate.conf` file ready to be consumed by the `logrotate` system daemon. This `logrotate` configuration will keep seven days of logs and will rotate them weekly.

The only thing left to do now is to set up `logrotate` to execute this configuration. It will suffice to place a soft link in the `/etc/logrotate.d` folder pointing to `logrotate.conf` generated file:

```
$ cd /etc/logrotate.d
$ ln -s /path/to/your/buildout/deployment.buildout/production/logrotate.conf
```

To make sure that the configuration is in place and well configured, we can execute the `logrotate` daemon in debug mode:

```
$ logrotate -d /etc/logrotate.d/logrotate.conf
```


Scheduling

Each operating system has its own task schedule facilities. For example, in Linux systems we can use **cron** to schedule tasks. As we said in previous sections, it's a good practice to schedule database backup every day. So we will setup a cron job in the `/etc/cron.daily` folder. It's enough to make a soft link to the backup script located in the `bin` buildout folder:

```
$ cd /etc/cron.daily
$ ln -s /path/to/your/buildout/deployment.buildout/bin/backup
```

This will trigger backup to be executed once a day.

Virtual hosting

Only under rare circumstances Zope is deployed without a reliable web server in front of it. For several reasons: flexibility and robustness are the most important. Flexibility because we can host several Zope servers in the same machine and do virtual hosting to access them, as long as we provide the other web resources and applications access from the front-end of the web server. Robustness, because it's better to have a reliable web server facing the exterior than Zope itself. Zope isn't security hardened against malicious users and it's easier to attack it (for example a Deny Of Service attack) than Apache, for example. Therefore, it's recommended not to expose it directly to the exterior.

There are a lot of reliable web servers out there, and all of them are configured the same way to do virtual hosting to a Zope server. I will recommend, **Apache** (<http://httpd.apache.org/>), although I had very successful experiences with **Nginx** (<http://wiki.nginx.org/>) lately.

Virtual hosting in Zope is done by rewriting (via the web server *rewrite* and *proxy* module) the incoming requests and passing them to the Zope server in proxy mode. We will have to enable these two modules in our web server. They are commonly used by web servers, and in some of them (such as Nginx) they are enabled by default.

VirtualHostMonster

This is a special Zope module used to ease the virtual hosting process. It's used to define Zope, how it has to rewrite all internal links to match the virtual host name used by the front-end web server. If this rewrite is not configured correctly, then all links published by Zope will be broken when we do virtual hosting. We have to specify how VirtualHostMonster has to do the link rewriting by defining it in the web server rewrite rule for that site.

Virtual hosting a root domain

This is an example of a virtual hosting from an Apache web server to a Zope server located in the same machine (`localhost`), listening to the 8301 port:

```
<VirtualHost *:80>
    ServerAdmin administrator@myplonesite.com
    ServerName myplonesite.com

    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>

    ErrorLog /var/log/apache2/error.log
    RewriteLog /var/log/apache2/rewrite.log

    LogLevel warn

    CustomLog /var/log/apache2/access.log combined

    RewriteEngine on
    RewriteOptions inherit

    RewriteRule ^/(.*) \
        http://localhost:8301/VirtualHostBase/\
        http/myplonesite.com:80/intranet/VirtualHostRoot/$1 \
        [P,L]

    ProxyVia On

</VirtualHost>
```

The highlighted line does the whole job. Notice the scaped newlines, it should be a single line directive. It defines a rewrite rule for the current example virtual host domain (`myplonesite.com`). All requests made to this domain will be rewritten:

```
RewriteRule ^/(.*)
```

They will be rewritten following the rest of the line convention. The request will arrive to Zope and `VirtualHostMonster` will interpret them and do the link rewriting. Let's dissect this line:

```
http://localhost:8301/
```

This part will tell Apache where to send the (proxied) request:

```
VirtualHostBase/http/myplonesite.com:80/intranet/
```

This part is for `VirtualHostMonster`. It defines the base URL of the link rewriting. Defined here also is the name of the Plone site (`intranet`):

```
VirtualHostRoot/$1 [P,L]
```

This last part tells `VirtualHostMonster` which is the root of the site relative to the base URL. The `$1` will tell the web server where to append what's remaining of the requested URL from the first `/` that appeared. This is issued by the previous `(. *)` regular expression. The modifiers at the end of the `RewriteRule` definition are to indicate that this rewrite has to be done in proxy mode and that the execution of rewrite rules must be stopped if matched.

Virtual hosting a domain subdirectory

Another common virtual hosting use case. Let's look at this example:

```
<VirtualHost *:80>
  ServerAdmin administrator@myplonesite.com
  ServerName myplonesite.com

  <Directory />
    Options FollowSymLinks
    AllowOverride None
  </Directory>

  ErrorLog /var/log/apache2/error.log
  RewriteLog /var/log/apache2/rewrite.log

  LogLevel warn

  CustomLog /var/log/apache2/access.log combined

  RewriteEngine on
  RewriteOptions inherit

  RewriteRule ^/intranet($|/.*) \
http://localhost:8301/VirtualHostBase/\
http://myplonesite.com:80/intranet/VirtualHostRoot/\
_vh_intranet$1 [P,L]

  ProxyVia On
</VirtualHost>
```

Again, it should be a single line directive. In this use case, the first part of the rewrite rule is slightly different to accommodate the desired subdirectory for our site. It will be available under this URL: `http://myplonesite.com/intranet`. The subdirectory name appears here, and, as rule of thumb, it has to match the last part of the line:

```
VirtualHostRoot/_vh_intranet$1 [P,L]
```

This will tell VirtualHostMonster to know that we are using a subdirectory of the base site URL. Notice that the `_vh_` prefix is needed for VirtualHostMonster to process the subdirectory.



RewriteRule witch

There is an online help application called RewriteRule Witch (<http://betabug.ch/zope/witch>). It provides an easy to fill form that will help to mount your VirtualHostMonster rewrite rule. In case of doubt, or as reference you can always ask the witch!!

Small intranet deployments

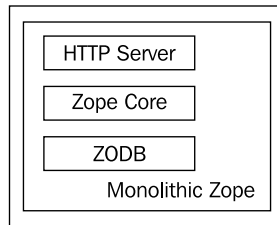
This is the simplest scenario possible. A site without great traffic requirements and a small number of potential users. For small deployments, the best approach is to go for a monolithic Zope. It requires less attention and work from the systems administrator point of view and it's the easiest to deploy.

Monolithic Zope

This is the simplest way to configure Zope; we have been using it since Chapter 2. It includes in a single server process:

- ZPublisher, the HTTP deliver engine with FTP, WebDAV, and XML-RPC capabilities
- ZServer, the core of Zope Application server
- ZODB database access server layer

These components are packed in what is called the Zope server or a monolithic Zope. From the operating system point of view, there is only a process involved in running Zope, and from the network application layer there is only one port listening.



Performance

This configuration is good enough for testing and developing purposes, and low requirements scenarios. The performance usually accomplished by a monolithic Zope is about five to six pages per second. This would allow up to four concurrent users at peak loads.

Scalability

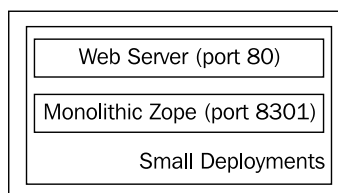
If the requirements of our intranet increases with the time, we can always change the type of deployment to accommodate the new ones. Switching between different kinds of deployments is not a complex procedure and often involves a simple reconfiguration of our buildout.

Buildout for small deployments

The buildout needed to build this type of deployment will be the buildout base configuration introduced in the previous section. It will provide us a monolithic Zope server, backup, log rotation and other basic capabilities.

Small deployments layer diagram

This is the resultant server layers diagram for small deployments:



Medium intranet deployments

This is the most common scenario. An intranet with mid-sized traffic and load requirements and an average number of intranet users. In this kind of deployment, you have to introduce cluster capabilities. We should use ZEO server as our database backend and a number of ZEO clients as frontend. The number of the ZEO clients will be determined by the requirements of our intranet.

ZEO (Zope Enterprise Objects)

ZEO is the load-balancing system used with Zope. ZEO server is a storage server that allows multiple Zope instances, called ZEO clients, to connect to a single database. In this scenario, all the database operations are delegated to ZEO, becoming the central database server for our deployment. ZEO clients will assume HTTP services, script and code execution and templates rendering.

Adding a ZEO server to our buildout

We are going to modify the basic buildout shown in the previous sections to accommodate a ZEO server. We should add this section to our `buildout.cfg` file:

```
[zeoserver]
recipe = plone.recipe.zope2zeoserver
zope2-location = ${zope2:location}
zeo-var = ${buildout:directory}/var
zeo-address = ${ports:zeo-server}
```

This buildout recipe will add a ZEO server in our deployment. We should, in consequence, modify the `ports` section too:

```
[ports]
...
zeo-server = 8100
```

And add this section to the `parts` directive in the `buildout` section:

```
[buildout]
parts = ...
       zeoserver
       ...
```

ZEO clients

We can have as many ZEO clients as our deployment would need. If we use more than one ZEO client to access our ZEO server, it will provide us service redundancy. If something ever happens to one of our ZEO clients, then the others will continue to respond to requests.

However, this will add some complexity to our deployment, because we will need an engine to distribute the load between all the ZEO clients we were using.

Scalability

Virtually, we can connect as many ZEO clients as we like to a ZEO server. The number will be driven by our site's requirements. ZEO clients and ZEO server are connected via the TCP protocol, so we can place them in different physical servers as long as a reliable network connects them.

For this reason, our deployment has a potential for infinite horizontal scalability, limited only by the hardware we might use. We can begin with one physical machine to accommodate both the ZEO servers and clients. If we need better performance, we can always increase it by growing horizontally by deploying more ZEO clients. These ZEO clients can be deployed in the same machine, or separate them in several machines to maximize performance.

Performance

Each ZEO client will give us approximately the same performance as a monolithic Zope server (2-4 page requests per second). So, it will depend on the number of ZEO clients deployed.

Adding ZEO clients to our buildout

We will use two recipes to declare them in `buildout.cfg`. We should add the following section:

```
[instance-settings]
eggs =
    Plone
    ${buildout:eggs}
zcml =
products = ${buildout:directory}/products
user = admin:admin
zodb-cache-size = 5000
zeo-client-cache-size = 300MB
```

```

debug-mode = off
zope2-location = ${zope2:location}
zoe-client = true
zoe-address = ${ports:zoe-server}
effective-user = ${users:zoe}

```

For each ZEO client that we want to add to our deployment, we should add a section, as the following, changing the name of the section and ports used by the ZEO client.

```

[instance1]
recipe = collective.recipe.zope2cluster
instance-clone = instance-settings
http-address = ${ports:instance1}
zope-conf-additional =
environment-vars =
    PYTHON_EGG_CACHE ${buildout:directory}/var/.python-eggs

```

Let's assume we are adding two ZEO clients to the deployment. Then, we have to adjust the `hosts` and `ports` sections accordingly:

```

[hosts]
...
instance1 = localhost
instance2 = localhost

[ports]
...
instance1 = 8301
instance2 = 8302

```

And add this section to the `parts` directive in the `buildout` section:

```

[buildout]
parts = ...
    instance1
    instance2
    ...

```

Finally, we should delete the existing monolithic Zope instance, by deleting the `instance` section and the line referring to this section in the `parts` directive of the `buildout` section.

Load balancer

We can use several techniques to achieve load balancing. We can use hardware load balancers (an expensive, but a very reliable option), or we can use load-balancing capabilities of our favorite web server (Apache and Nginx have this capability). You can use the one you are more familiar with.

However, we are going to cover how to realize this task with a dedicated software load balancer server called **HAproxy**. It's a free, very fast, and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications.

For adding it to our deployment buildout, we should add these lines to our `buildout.cfg` file:

```
[haproxy-build]
recipe = plone.recipe.haproxy
url = ${downloads:haproxy}
cpu = ${build:cpu}
target = ${build:target}
```

This recipe will download and build HAproxy. As we've already noticed, this section relies on other buildout sections that we haven't set up yet. Let's do so by adding this section. Modify it according to our machine specifications, if applicable:

```
[build]
cpu = i686
target = linux26
```

Then add the `downloads` section. This will serve for grouping URL downloads for later use in other buildout sections:

```
[downloads]
haproxy = http://haproxy.1wt.eu/download/1.4/src/haproxy-1.4.0.tar.gz
```

Once HAproxy is built, then we should specify how to configure it. We can do it by adding this new section:

```
[haproxy-config]
recipe = collective.recipe.template
input = ${buildout:directory}/templates/balancer.conf.template
output = ${buildout:directory}/production/balancer.conf
group = plone
maxconn = 24000
```

This section will use the buildout template recipe for generating a configuration file for HAproxy from the template file called `balancer.conf.template` located in the `templates` folder. The resultant configuration file (`balancer.conf`) will be placed in the `production` folder. This file will configure the most important aspects of the balancer settings:

```
global
  log 127.0.0.1 local6
  user ${users:balancer}
  group ${haproxy-config:group}
  maxconn ${haproxy-config:maxconn}
  ulimit-n 65536
  daemon
  nbproc 1

defaults
  mode http
  option httpclose
  option abortonclose
  retries 3
  option redispatch
  monitor-uri /haproxy-ping

  timeout connect 7s
  timeout queue 300s
  timeout client 300s
  timeout server 300s

  stats enable
  stats uri /haproxy-status
  stats refresh 5s
  stats realm Haproxy\ statistics

frontend myIntranet-Balancer
  bind ${hosts:balancer}:${ports:balancer}
  default_backend myIntranetBackend

  capture cookie __ac len 10
  option httplog
  log 127.0.0.1 local6

# Load balancing over the zope instances

backend myIntranetBackend
```

```
    appsession __ac len 32 timeout 1d
    balance roundrobin
    cookie serverid insert nocache indirect
    option httpchk GET /

    server zope8301 ${hosts:instance1}:${ports:instance1} cookie z8301
    check maxconn 1 maxqueue 2 rise 1
    server zope8302 ${hosts:instance2}:${ports:instance2} cookie z8302
    check maxconn 1 maxqueue 2 rise 1
```

The highlighted lines are the most important of this file. They define the port for the balancer and the backends it will use. All requests made to the balancer port (8310) will be balanced between the two zope instance ports (8301 and 8302).

We have to add balancer to the `hosts`, `ports`, and `users` sections:

```
[hosts]
...
balancer = localhost

[ports]
...
balancer = 8310

[users]
...
balancer = plone
```

Finally, add `haproxy-build` and `haproxy-config` to the `parts` directive section, as shown next:

```
[buildout]
parts = ...
       haproxy-build
       haproxy-config
       ...
```

When HAProxy is running, we can access a status page from the URL:
<http://localhost:8310/haproxy-status>



For more information on HAProxy configuration settings and advanced features: <http://www.haproxy.org/>.

Supervisor to rule them all

Now that we have so many servers, clients, and helper processes, we need a little help to control all of them from one central location. We are going to introduce a very useful tool called Supervisor (<http://supervisord.org/>). Supervisor is a client/server system that allows its users to monitor and control a number of processes on UNIX-like operating systems. Setting it up in buildout is easy with the appropriate recipe. Just add this section to your `buildout.cfg` file:

```
[supervisor]
recipe = collective.recipe.supervisor
port = ${ports:supervisor}
user = admin
password = admin
serverurl = http://${hosts:supervisor}:${ports:supervisor}
programs =
  10 zeo ${zeo:location}/bin/runzeo true plone
  20 instance1 ${buildout:directory}/parts/instance1/bin/runzope true
    ${users:zope}
  20 instance2 ${buildout:directory}/parts/instance2/bin/runzope true
    ${users:zope}
  40 haproxy ${buildout:directory}/bin/haproxy [ -f
    ${buildout:directory}/production/balancer.conf -db ]
```

The `programs` directive will tell Supervisor what processes it should take care of. Here we should declare all the processes to be launched by this buildout. Since Supervisor uses a port and a hostname, we should add these to `hosts` and `ports` sections:

```
[hosts]
...
supervisor = localhost

[ports]
...
supervisor = 9001
```

Finally, add `supervisor` to the `parts` directive buildout section.

```
[buildout]
parts = ...
       supervisor
       ...
```

Using Supervisor

Supervisor is very easy to use. All we have to do is launch the supervisor daemon, and it will launch all managed processes. We can launch Supervisor by executing this command line:

```
$ ./bin/supervisord
```

By default, once Supervisor is launched, it will launch all managed processes. We can see the status of all of them by executing:

```
$ ./bin/supervisorctl status
```

Or directly access to the interactive prompt:

```
$ ./bin/supervisorctl
```

The command line has these main commands:

- `stop [process]`
- `start [process]`
- `restart [process]`
- `status`: It shows the status of all processes and the running time of each of them
- `tail [process]`: It shows the default `stdout` of the process
- `fg [process]`: Connects to a process in foreground mode

We can access the rest of the commands by executing the `help` command.

Modifying the web server settings

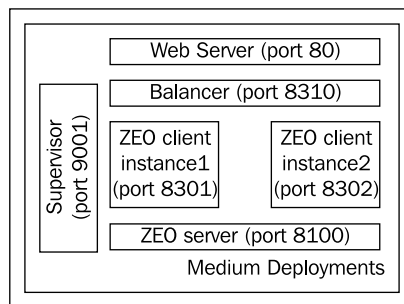
Now we should close the circle by setting up our web server. We should modify the port of the rewrite rule to point to the balancer:

```
RewriteRule ^/intranet($|/.*) http://localhost:8310/VirtualHostBase/  
http/myplonesite.com:80/intranet/VirtualHostRoot/_vh_intranet$1 [P,L]
```

We still need the full VirtualHostMonster formatted rewrite rule, of course. Now, instead of the port of the previous single instance, we will send all requests to the port 8310 where the balancer is listening.

Medium deployments layer diagram

Again, here is the updated service layer diagram for medium deployments:



Large intranet deployments

Finally, we move onto the serious issues. Large intranet scenarios will require us to deploy the best available artillery. High traffic loads, high number of concurrent users, and more than a hundred users will require us to tweak and improve each component of our system architecture. Different problems may arise in this scenario, like managing lots of users, outage of hardware resources (usually memory), saturation of ZEO clients, and so on. If we experiment with some of this, we are in front of a large scenario.

Adding cache to our deployment

This is the first countermeasure in order to decrease the saturation of ZEO clients. The purpose of a cache server is to capture as many requests as possible before they reach the backend servers. It consists of two parts: the inclusion of a helper application in Plone called **CacheSetup**, and using a cache server.

Products.CacheSetup add-on product

Formerly known as CacheFu, it speeds up Plone sites transparently using a combination of memory, proxy, and browser caching. Although it can be used as standalone, the best results are accomplished coupling it with a cache server such as Apache, Squid, or Varnish.

The purpose of CacheSetup is to add an intelligent layer between the client browser, or the cache server, and the Plone site. Its task is to inform the client browser or the cache server if a particular page requested must be refreshed from the cache. This action is called **purging**. It will add the necessary HTTP headers to do the job.

Install CacheSetup as any other add-on product:

```
[buildout]
...
eggs = ...
      Products.CacheSetup
```

Then install it from Plone's control panel, Add-on products configlet. A new configlet will be available called **Cache Configuration Tool** in the control panel. The configlet is composed of several configuration forms and folders containing special objects. These objects represent cache policies and rules for our site.

CacheSetup is disabled by default; we have to enable it and fill the form in located in `main` tab:

main	policies	rules	headers	memory	actions ▼	add cache policy
------	----------	-------	---------	--------	-----------	------------------

Cache Configuration Tool

▲ Up to Site Setup

Enable CacheFu
 Uncheck to turn off CacheFu's caching behavior. Note: Disabling CacheFu does not purge proxy or browser caches so stale content may still continue to be served out of those caches.

Active Cache Policy
 Please indicate which cache policy to use.

With Caching Proxy
 Without Caching Proxy

Proxy Cache Purge Configuration ■
 If you are using a caching proxy such as Squid or Varnish in front of Zope, CacheFu needs to be able to tell this proxy to purge its cache of certain pages. If Apache is in front of Squid/Varnish, then this depends on Apache's "virtual hosting" configuration. The most common Apache configuration generates VirtualHostMonster-style URLs with RewriteRules/ProxyPass. If you have a legacy CacheFu 1.0 Squid-Apache install or other custom Apache configuration, you may want to choose the "custom URLs" option and customize the rewritePurgeUrls.py script.

Purge with VHM URLs (squid/varnish behind apache, VHM virtual hosting) ▼

Site Domains
 Enter a list of domains for your site. This is not needed if you chose "No Purge" under the Proxy Cache Purge Configuration option above. If your site handles both http://www.mysite.com:80 and http://mysite.com:80, be sure to include both. Also include https versions of your domains if you use them. Be sure to include a port for each site.

http://myplonesite.com:80

Proxy Cache Domains
 Enter a list of domains for any purgeable proxy caches. This is not needed if you chose "No Purge" or "Simple Purge" under "Proxy Cache Purge Configuration" above. For example, if you are using Squid with Apache in front, there will commonly be a single squid instance at http://127.0.0.1:3128

http://localhost:8001

Compression
 Should Zope compress pages before serving them, and if so, what criteria should be used to determine whether pages should be gzipped? The most common settings are "Never" (no compression) or "Use Accept-Encoding header" (only compress content if the browser explicitly declared support for compression).

Use Accept-Encoding header ▼

Vary Header
 Value for the Vary header. If you are using gzipping, you may need to include "Accept-Encoding" and possibly "User-Agent". If you are running a multi-lingual site, you may also need "Accept-Language". Values should be separated by commas. (Upon submit, this value will be cleaned up and checked for any obvious omissions)

Accept-Encoding

save cancel

The following is a summary of the available options:

- **Active Cache Policy:** It is used to define if a cache server is used or not
- **Proxy Cache Purge Configuration:** If we are using a cache server, indicate here the type
- **Site Domains:** It lists the domains used by the site
- **Proxy Cache Domains:** A list of domains for any purgeable cache. We should indicate here the host names and the ports for our cache server instances.

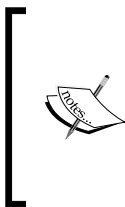
CacheSetup is configured out-of-the-box with the optimal settings for use in the most common scenarios. However, we can modify the cache rules to tweak the cache settings to satisfy our requirements. There are specific rules for each content type use-case and others for special cases like RSS or DTML files. Setting up CacheSetup is very important because it's the glue part for using a cache system correctly.

Cache server

A cache server, also known as a reverse proxy, dispatches in-bound requests to a backend server (or set of servers) featuring backend response caching and HTTP accelerator capabilities. The idea is to free ZEO client backends from repetitive requests: non-dynamic content, dynamic but little modified content, or theme elements such as CSS files, images, and JavaScripts.

We can find several cache server products out there. We are going to choose Varnish (<http://varnish-cache.org/>) for the job. Varnish is a state-of-the-art, high-performance HTTP accelerator. It uses some advanced features in Linux 2.6, FreeBSD 6/7 and Solaris 10 to achieve its high performance.

Varnish has its own configuration language called VCL. The VCL language is a small domain-specific language designed to be used to define request handling and document caching policies for Varnish.



Cache servers are complex pieces of software. Because of that, configuring them is not trivial. However, we are going to cover how to install Varnish using buildout and how to configure it for a standard Zope/Plone use case. If you want to tweak this configuration, feel free to read more about Varnish and the VCL language at: <http://varnish-cache.org/wiki/VCL>.

Building and configuring Varnish

Add this new section to our `buildout.cfg` file:

```
[varnish-build]
recipe = hexagonit.recipe.cmmi
url = ${downloads:varnish}
```

This will build Varnish from sources. We will have to add this new line referring to the Varnish download URL to the `downloads` section:

```
[downloads]
...
varnish = http://sourceforge.net/projects/varnish/files/varnish/2.0.6/
varnish-2.0.6.tar.gz/download
```

We will use a recipe to configure it, and another to compile a configuration template file:

```
[varnish]
recipe = plone.recipe.varnish
daemon = ${buildout:directory}/parts/varnish-build/sbin/varnishd
mode = foreground
bind = ${hosts:cache}:${ports:cache}
cache-size = 1G
user = ${users:cache}
config = ${buildout:directory}/production/cache.conf
telnet = ${hosts:cache}:${ports:cache-telnet}

[varnish-config]
recipe = collective.recipe.template
input = ${buildout:directory}/templates/cache.conf.template
output = ${buildout:directory}/production/cache.conf
```

Varnish process binds to the port specified by the `bind` directive. Here we configure cache size, owner user, and the telnet port. We set the cache size to 1GB, although we can adjust it depending on how much memory we have in our machine. Varnish has a Telnet-like managing console. We can connect to it by using any Telnet compatible program against the specified port. The rest of the configuration is done using an external configuration file generated by a buildout template.

We should modify `users`, `hosts`, and `ports` sections to set up the new process:

```
[hosts]
...
cache = localhost

[ports]
...
cache = 8001
cache-telnet = 9002

[users]
...
cache = plone
```

We should add these lines to the `parts` directive of the `buildout` section too, in order to make the modifications effective:

```
[buildout]
parts = ...
    varnish-build
    varnish
    varnish-config
    ...
```

Finally, add Varnish to be controlled by Supervisor. Add this line to the `supervisor` section:

```
[supervisor]
50 varnish ${buildout:directory}/bin/varnish true ${users:cache}
```

Default VCL configuration template file

It's time for a brief review of the VCL configuration file included in the available support code for this chapter. However, it's a long file, so we are going to show only the most interesting parts of it. The file, `varnish.conf.template`, is located in the `templates` folder.

The first two lines will define the backend. We are going to place the cache server between the web server and the balancer, so the cache server backend should be the balancer:

```
/* Configure balancer server as back end */
backend balancer {
    .host = "${hosts:balancer}";
    .port = "${ports:balancer}";
}
```

This will allow only to perform a cache purge from the localhost:

```
/* Only allow PURGE from localhost */
acl purge {
    "localhost";
}
```

VCL defines a workflow for handling incoming requests. Each request follows this workflow from the beginning until it's processed, and the response is sent to the client. The requests flow through the workflow states (*subroutines* in VCL jargon). These subroutines can be extended from VCL to tweak its default functionality.

On arrival, the `vcl_recv` subroutine will be executed. In this subroutine, we will define the backend:

```
sub vcl_recv {

    /* Send to backend upon receive */

    set req.grace = 120s;
    set req.backend = balancer;
```

Finally, it will lookup in the existing cache:

```
lookup;
}
```

If the requested URL is found in the cache, then the `vcl_hit` subroutine is processed:

```
sub vcl_hit {
    if (req.request == "PURGE") {
        purge_url(req.url);
        error 200 "Purged";
    }

    if (!obj.cacheable) {
        pass;
    }
}
```

If the requested URL is not found, then `vcl_miss` subroutine is processed:

```
sub vcl_miss {
    if (req.request == "PURGE") {
        error 404 "Not in cache";
    }
}
```

And the object is fetched from the backend and stored in the cache. The subroutine `vcl_fetch` is responsible for this. Somehow, we will add some actions to this subroutine:

```
sub vcl_fetch {
    set obj.grace = 120s;

    if (!obj.cacheable) {
        pass;
    }
    if (obj.http.Set-Cookie) {
        pass;
    }
    if (obj.http.Cache-Control ~ "(private|no-cache|no-store)") {
        pass;
    }
    if (req.http.Authorization && !obj.http.Cache-Control ~ "public")
    {
        pass;
    }
}
```



For more information about the VCL workflow refer to:
<http://varnish-cache.org/wiki/VCLExampleDefault>

Modifying the web server settings

Once we have the cache server in place, we should update our web server configuration. We should modify the port of the rewrite rule to point to the cache server:

```
RewriteRule ^/intranet($|/.*) http://localhost:8001/VirtualHostBase/  
http/myplonesite.com:80/intranet/VirtualHostRoot/_vh_intranet$1 [P,L]
```

Instead of the port of the previous single instance, now we will send all the requests to the 8001 port where the cache server is listening.

Spanning services in separate servers

Horizontal scaling is the best option if we have squeezed out all the computing resources of our server. We can separate our system processes, each one in dedicated hardware. For example, separate the pure backend ZEO server from the frontend part composed by the ZEO clients, web server, and balancer. If necessary, we can also split the frontend processes again to gain more performance.

However, monitoring our server resources before investing in new hardware is a good idea. Just make sure that our resources are being used at their best, and then make the decision.

Increasing the ZEO client instances

Either on the same hardware or on separate ones, increasing ZEO clients will increase our site's performance. We just have to modify our buildout accordingly. Simply copy any instance section, and modify the name of the section and the `http-address` directive. For example, following are the instances 3 and 4:

```
[instance3]
recipe = collective.recipe.zope2cluster
instance-clone = instance-settings
http-address = ${ports:instance3}
zope-conf-additional =
environment-vars =
    PYTHON_EGG_CACHE ${buildout:directory}/var/.python-eggs
```

```
[instance4]
recipe = collective.recipe.zope2cluster
instance-clone = instance-settings
http-address = ${ports:instance4}
zope-conf-additional =
environment-vars =
    PYTHON_EGG_CACHE ${buildout:directory}/var/.python-eggs
```

Update the `ports` section:

```
[ports]
...
instance3 = 8303
instance4 = 8304
```

Also update the `parts` directive:

```
[buildout]
...
parts = ...
    instance3
    instance4
    ...
```

Updating balancer configuration

If we add more ZEO clients, we have to update our balancer configuration in order to set up the new backends:

```
server zope8303 ${hosts:instance3}:${ports:instance3} cookie z8303
check maxconn 1 maxqueue 2 rise 1
    server zope8304 ${hosts:instance4}:${ports:instance4} cookie z8304
check maxconn 1 maxqueue 2 rise 1
```

Setting LDAP as an external user database

In order to manage a very large database of users, it's always advisable to use an external and probably already existing user repository, such as an LDAP directory compatible server. **LDAP (Lightweight Directory Access Protocol)** is a standard application protocol for querying and modifying data using directory services running over TCP/IP. If you are in a large organization or company, it is probable that you already have an LDAP compatible directory. For example, Microsoft Active Directory, Novell, Sun One, and others are LDAP compatible. Maybe you are already using the Open Source implementation (OpenLDAP) to authenticate your applications.

We can use our existing LDAP compatible directory as source for our intranet users. By doing this, we will give access to the intranet with the username and password of the LDAP directory. We can even use LDAP stored groups to manage the site's security.

There is a great add-on product called `plone.app.ldap` that will help us to configure the connection. It consists of a helper application and the Python modules required.

We can install `plone.app.ldap`, as any other add-on product:

```
[buildout]
...
eggs = ...
    plone.app.ldap
```

```
[instance]
...
zcml = ...
    plone.app.ldap
```

Pin the version of `plone.app.ldap` to 1.1. We can do this by using the `versions.cfg` file, as shown previously in this chapter:


```
[versions]
...
plone.app.ldap = 1.1
```

Install it from the add-on products configlet. It will install a new configlet called **LDAP configuration** in the control panel.

Connecting our intranet to an LDAP compatible server is not a big deal. We only need to know some information regarding our LDAP server and fill in some other relevant data:

- **LDAP server type:** This will let us choose if our directory server is Active Directory or LDAP based.
- **rDN attribute:** This attribute is used to uniquely identify our directory users. It is used to build the **distinguished name (DN)** for users that are being created in our LDAP directory. This is commonly either the users full name (`cn` property) or the **userid** (`uid` property).
- **user id attribute:** This attribute is used as the userid inside Plone for LDAP users. It has to be unique for all users.
- **login name attribute:** This attribute is used as the login name for the LDAP users logging into our site. In most cases, this should be the same as the user id attribute.
- **LDAP object classes:** Each of the objects in the LDAP database have a structural object class, and optionally several supplemental object classes. These classes define the required and optional properties that can be present on an object. Classes can be entered in a comma separated list. For LDAP servers, user classes are derived from the *top* and *person* classes. Specify them separated by colons.
- **Bind DN:** This the DN of a manager account in the LDAP directory. This must be allowed to access all user and group information, as well as be able to update and create users and groups. Please note that Plone only supports simple binds. SASL is not supported. This user shouldn't be an LDAP manager.
- **Bind password:** This is the password to use when binding to the LDAP server.

- **Base DN for users:** This is the location in our LDAP directory where all the users are stored.
- **Search scope for users:** The search scope determines where the LDAP server will search for users. With **BASE**, it will only look for users who are directly in the user base location. **SUBTREE** will allow the server to also look in subfolders of the user base location.
- **Base DN for groups:** This is the location in your LDAP directory where all the groups are stored.

 Don't be overwhelmed by the number of settings to be configured! If you don't know some of them, or you are not familiar with them, you may ask your systems administrator about them.

We have to add a server in the **LDAP server** tab. Click the **Add LDAP server** button to access the server configuration form:

Add Server

Add an new LDAP or ActiveDirectory server.

Configure server

Enabled
Use this LDAP server

LDAP server ■
The address or hostname of the LDAP server.

LDAP connection type ■

Connection timeout ■
The timeout in seconds to wait for a connection to the LDAP server to be established.

Operation timeout ■
The timeout in seconds to wait for an operation such as a search or update to complete. If no timeout should be used use -1 as value.

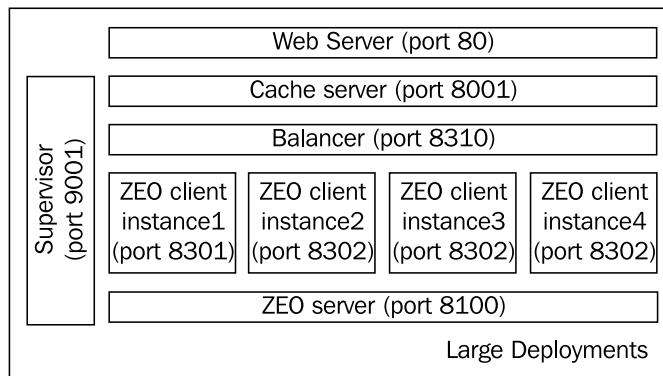
Specify the required settings:

- **LDAP server:** The address or host name of our server.
- **LDAP connection type:** Choose between simple LDAP, LDAP over SSL, or LDAP over IPC.
- **Connection timeout:** The timeout in seconds to wait for a connection to the LDAP server to be established.
- **Operation timeout:** The timeout in seconds to wait for an operation, such as a search or an update to complete. It's recommended to configure it to a value since the default is infinite.

Once we have configured all these settings, our LDAP directory users should be able to log into our intranet. Our local users should be able to login too.

Large deployments service layer diagram

This is the updated service layer diagram:



Summary

This chapter covered how to deploy our intranet, based on its capacity needs. We have learned how to use clustering in Zope, via ZEO servers and clients. You also know about how to configure:

- Backups, restores, log rotation, and ZODB packing
- Virtual hosting in Zope with VirtualHostMonster
- Rewriting in Apache
- ZEO and ZEO clients
- A load balancer (HAproxy)
- A cache server (Varnish)
- LDAP authentication

I hope you have enjoyed this book as much as I did writing it, and that it has helped you to build a well designed, useful and successful intranet. However, this is only the beginning, there's a lot of things that Plone can do for you. They are waiting for you to explore them.

Index

Symbols

`./bin/backup` script 249
`./bin/restore` script 249
`./bin/snapshotbackup` script 249
`./bin/snapshotrestore` script 249
`.tgz` file 25

A

`acl_users` 86
`acl_users` tool 121
administration tasks
 about 248
 database, backing up 249
 database packing 250
 database, restoring 249
 log files, rotating 250, 251
 scheduling 252

Apache

URL 252

ARFilePreview

features 174
installation 174

AROfficeTransforms

additional softwares 175
installation 174

asset library 9

B

blogs

about 162
configuring 165
Quills 162
Scrawl 165

buildout. *See* `zc.buildout`

buildout base configuration

buildout folder ownership, modifying 248
extended configuration, caching 246
newest directive, using 246
ports name sections, adding 247
process owners section, adding 247
versions file, adding 245, 246

buildout.cfg file 71

buildout.cfg, setting up

about 32
buildout section 33
extends directive 33
instance section 35
zope2 section 34, 35
zopepy section 36

buildout deployment

about 244
base configuration 245

buildout file 73

buildout section, buildout.cfg

about 33
extends directive 33
find-links directive 34
versions.cfg file 33

C

cache server

default VCL configuration file 270-272
Varnish 268
Varnish, building 269, 270
Varnish, configuring 269

CMF 7

CMS 8

collection

- about 138
- configuring, in edit mode 139
- creating 139, 140
- criteria, adding 140
- results, ordering 140
- using 138

collective.workflowed 93

community workflow

- draft 100
- for folders 100
- pending 100
- private 100
- published 100
- state diagram 100

community workflow, for folders

- draft 101
- private 101
- published 101
- state diagram 100

configlets 56

content

- folder, adding 47, 48
- managing 51
- metadata, setting 50
- page, adding 49
- settings 50
- standard Plone content types 49

content_icon file 192

Content Management System. See CMS

content management tabs, Plone

- about 45
- content actions 45
- content tabs 45
- content views 45
- default content actions 46

content rules

- about 199
- accessing 200
- actions, executing 203
- assigning, to folderish content types 204
- awareness, enabling 204
- defining 200
- events 202
- managing 200
- new rule, adding 201, 202

content structure

- about 46
- Plone site, default objects 47

control panel, ZMI

- about 66
- database management 67
- product management 68
- translation service 68

create command 20

CSSManager

- basic theming properties 232
- CSS customization, base_properties
 - used 232
- installing 230
- logged-in tab' look, changing 232
- using 231

cssregistry.xml file 241

CSS resource

- authoring.css.dtml 238
- base.css.dtml 238
- columns.css.dtml 238
- controlpanel.css.dtml 239
- deprecated.css.dtml 239
- forms.css.dtml 239
- invisibles.css.dtml 239
- member.css.dtml 238
- navtree.css.dtml 239
- ploneCustom.css.dtml 239
- portlets.css.dtml 239
- print.css.dtml 239
- public.css.dtml 238
- RTL.css.dtml 239

D

default views

- displaying 52

Dexterity

- about 196
- key features 196

directory structure, zc.buildout

- bin 31
- bin/buildout 31
- bin/instance 31
- bin/repozo 31
- bin/zoepypy 31
- buildout.cfg 31

- buildout.cfg, setting up 32
- downloads 31
- eggs 31
- parts 31
- products 31
- src 32
- var 32
- var/filestorage 32
- var/filestorage/Data.fs 32
- var/log/instance.log 32
- var/log/instance-Z2.log 32

discussion board

- PloneBoard 167

distinguished name. *See* DN

DN 275

document files, managing

- about 174
- ARFilePreview 174
- AROfficeTransforms 174
- OpenXML 175

Document Template Markup Language.

See DTML

DTML 232

dynamic intranet 199

E

EMS

- about 8
- WMS, differentiating 10

Enable External Editor feature attribute 58

Enable link integrity checks attribute 57

Enterprise Management Systems. *See* EMS

ESMTP 57

example intranet workflow, building

- draft state 126
- intranet state 126, 127
- private state 125
- state diagram 124
- state, requirements 124
- transitions, adding 127-129

Exclude from navigation setting 50

Expansion Template Attribute Language.

See METAL

Expose sitemap.xml.gz in the portal root attribute 58

Extended SMTP. *See* ESMTP

external edition

- about 211, 212
- enabling 213
- helper software, modifying 213
- installing 212

external edition installation

- in Linux 212
- in Mac OS X 213
- in Windows 212

extranets 15

F

factorytool.xml file 193

fg [process] command 264

field types, PloneFormGen

- Checkbox 159
- Date/Time 159
- Decimal Number 159
- File 159
- Label 159
- Lines 159
- Multi-Select 159
- Password 159
- Rich Label 159
- Rich Text 159
- Selection 159
- String 159
- Text 159
- Whole Number 159

Flash

- adding, to site's content 62

form generators

- about 157
- PloneFormGen 157

G

GenericSetup

- about 185, 186
- content types, closing 190-192
- importing, from product 189
- particular product profile, exporting 188
- particular product profile, importing 188
- snapshots 186
- snapshots and product profiles, comparing 188

using, for theme customization 241
XML files 186

Generic Setup, files

cssregistry.xml 241
jsregistry.xml 241
skins.xml 241
viewlets.xml 241

global roles

about 118
using 119

GloWorm

installing 228
using 229, 230

GloWorm Inspector Panel 229

GNU General Public License. *See* GPL

GNU Privacy Guard. *See* GPG

GPG 160

GPL 7

groups

administering, ZMI used 88
Group Properties tab 85
managing 85

H

HAproxy 260

header, layout structure

breadcrumbs, elements 43
logo, elements 43
personal bar, elements 43
portal tabs, elements 43
search box, elements 43
site action, elements 43

I

installation script, running

tasks 26

installing

CSSManager 230
external edition 212
GloWorm 228
installingPlone 18
OpenXML 175
Plone4ArtistsCalendar 152
PlonePopoll 170
PloneSurvey 172
vs.event 154, 155

intranet

calendering events 152
intranetcontent, adding 47
intranetfeatures 10
intranetprivate content, managing 129
intranetworkflow, choosing 123
intranetworkgroups, enabling 11

intranet deployments

administration tasks 248
buildouts deployment 244
large deployments 265
medium deployments 257
small deployments 255
virtual hosting 252

intranet information architecture, designing

category, types 137, 138
content, categorizing 137
important points 137
second levels 137

intranet workflow

about 102
externally visible state 102
for folders 103
internal draft state 102
internally published state 102
pending review state 102
private state 102

intranet workflow, for folders

internal draft state 103
private state 103

J

JavaScript for web statistics support

attribute 58

jsregistry.xml file 241

K

keywords, category type 137

Kupu's HTML view button 144

L

large deployment

balancer configuration, updating 274
cache configuration tool, options 268
cache server 268

- CacheSetup, adding 266
- LDAP setup, as external user
 - database 274-277
- Products.CacheSetup add-on product 266
- service layer diagram 277
- services, spanning in separate services 273
- web server settings, modifying 272
- ZEO client instances, increasing 273

layout structure, Plone

- columns 44
- contents 44
- contents, elements 44
- footer 44
- header 43
- regions 42

LDAP 274

Lightweight Directory Access Protocol. *See* LDAP

LinguaPlone 63

Linux

- about 25
- external edition, installing 212

LiveSearch 43

local roles

- about 118
- using 119

logrotate daemon 250

M

Mac OS X

- about 27
- external edition, installing 213

Macro Expansion Tag Attribute Language .
See METAL

medium deployments

- layer diagram 265
- load balancer 260-262
- service layer diagram 265
- Supervisor 263
- Supervisor, using 264
- web server settings, modifying 265
- ZEO 257

METAL 95, 222

mingw32

- URL 19

monolithic Zope

- about 255
- performance 256
- scalability 256

N

nasty tags 62

new add-on products, installing

- via buildout 71-74
- Zope 2 add-on product 75

next previous navigation

- enabling 142

Nginx

- URL 252

O

OpenXML

- installing 175
- requirements 176

out-of-box workflows

- about 98
- community workflow 100
- intranet workflow 102
- one state workflow 101
- simple publication workflow 99

own custom product, building

- about 178
- egg, creating 179
- naming 178
- Plone product egg, anatomy 180
- ZCML configuration files 183

own theme add-on product, building

- CSS debug mode, enabling 236
- CSS files, customizing 238
- CSS files, resetting 239
- logo image, customizing 236
- new logo image, adding 236
- plone.logo viewlet, customizing 237
- product, installing 234, 235
- site logo, customizing 236
- skin layer resources, customizing 236
- steps 233

P

P4A 152

page rendering, Plone

- about 217
- folder 217
- main_template.pt page template, rendering 225-227
- page, composing 225
- skins layers 221
- skins tool 218, 220
- sub folder 217
- workflow diagram 216, 217
- Zope page templates 221

PAS

- about 77
- components 87
- feature 87

paste command-line tool 29

PasteScript 29

path separators, Windows

- backslashes (\) 18
- command prompt(>) sign 18
- forward slashes (/) 18

PHP For Applications. See P4A

Placeless Translation Service. See PTS

Plone

- about 12
- benefits 8, 10
- blogs 162
- building 8
- collection 138
- comparing, with CMS solutions 8-11
- content management tabs 45
- control panel 56
- feature 58, 59
- front page view, anonymous users 45
- front page view, logged in users 45
- GenericSetup 185
- installing 18
- layout structure 41
- page rendering 216
- PAS 87
- Plone 4, features 18
- resource registries 227
- security entities 78
- setting up 20-24

- Sharing tab 96
- site accessing, URL 38
- standard folder views 52
- syndication 205
- tree-based hierarchy 11
- unified installer 24
- versions, Plone 3 18
- versions, Plone 4 18

Plone 4

- features 18

Plone4ArtistsCalendar

- about 152
- calendar activated folder 153
- features 152, 153
- installing 152

PloneBoard, discussion board

- about 166
- forums, content type 168
- installing 167
- message board content type 168
- permission adjustments, for intranet users 168, 169
- working 168

Plone community

- about 14
- blogs 15
- chatrooms 15
- Plone Conference 15

Plone Controller 25

Plone control panel

- about 56
- accessing, Site Setup action 56
- add-on products 60
- components, add-on configuration 56
- components, Plone configuration 56
- configlets 56
- content rules 61
- content types 59
- default language 63
- errors 62
- groups 58
- HTML filtering 62
- mail control panel 57
- maintenance 61
- markup 63
- navigation 64
- search settings, refining 65

- security aspects, configuring 59
- site 57
- site, attributes 58
- theme 65
- users 58
- wiki formatting 64
- PloneFormGen, form generators**
 - action adapters 160
 - CAPTCHA, integrating 161, 162
 - content types 160
 - dependencies 158
 - extensibility 161
 - field types 159
 - installing 157
 - third-party products 161
 - working 158
- Plone, installing**
 - versions, requiring 19
 - zc.buildout, requirements 19
- Plone pluggable authentication service.** *See* **Plone PAS**
- PlonePopoll**
 - about 170
 - default permissions, adjusting 171
 - installing 170
 - local roles, using 171
 - working 170
- Plone product egg, anatomy**
 - egg documentation files 180
 - egg setup files 181, 182
 - main content products 182, 183
- Plone site**
 - new add-on products, installing 71
 - root objects 70
 - ZMI view 69
- Plone site, root objects**
 - acl_users 70
 - error_log 70
 - mailhost 70
 - members 70
 - portal_actionicons 70
 - portal_actions 70
 - portal_catalog 70
 - portal_css 70
 - portal_factory 70
 - portal_javascript 70
 - portal_languages 70
 - portal_memberdata 70
 - portal_membership 70
 - portal_migration 70
 - portal_properties 70
 - portal_quickinstaller 70
 - portal_setup 71
 - portal_skins 71
 - portal_transforms 71
 - portal_types 71
 - portal_workflow 71
- Plone's Unified Installer.** *See* **PUI**
- PloneSurvey**
 - about 172
 - default permissions, adjusting 173
 - installing 172
 - working 172
- Plone UI**
 - roles, adding to 131
- Pluggable Authentication Service.** *See* **PAS**
- portal_javascript tool 228**
- portal_setup tool 188**
- portal_view_customizations tool 230**
- portal_workflow tool 123**
- portlets**
 - default portlets 53, 54
 - managing 53
- presentation mode**
 - about 142
 - enabling 143
 - heading styles 143
 - leading slide 143
 - slide, formatting 144
- private content**
 - managing 129, 130
- private sections**
 - creating 130
- product**
 - add-on product install section 185
 - buildout 184
 - using, for security configuration 193
- Products.DCWorkflowGraph**
 - about 109, 110
 - installing 109
- product, using for security configuration**
 - existing workflow, modifying 194, 195

- role map assignment to permissions, defining 193
- workflow, creating 194, 195

PTS 63

public websites 15

publishtointranet 127

PUI 24

PyPI

- accessing, URL 28

Python

- about 12
- eggs 28
- features 12
- version 19
- Win32 extensions 19

Python Imaging Library

- URL 19

Python Package Index. *See* **PyPI**

Q

Quills, blogs

- about 162
- features 163
- installing 163
- main view 164
- portlets 164

R

RDBMS 13

Relational Database Management System.

See **RDBMS**

resource registries

- about 227
- CSS resource registry 227
- JavaScript resource registry 228

restart [process] command 264

reverse proxy. *See* **cache server**

RewriteRule witch 255

role policy

- additional manager users, creating 121
- administrator users, creating for Zope instance 121
- local roles, delegating 122
- Manager restricted role permissions, granting 122
- Manager role use, restricting 120, 121

- non-managers administer local roles, allowing 123
- requirements 120

roles

- about 79
- administering, ZMI used 88
- anonymous 79
- authenticated 79
- contributor 79
- editor 79
- global roles 80
- local roles 80
- manager 79
- member 79
- owner 79
- reader 79
- reviewer 79
- rolesabout 118
- rolesassigning 118
- roleslocal roles 118
- rolesuser roles 118

roles, adding to Plone UI

- collective.sharingroles, using 133
- custom product, using 132

S

S5 JavaScript library 142

Scrawl, blogs

- features 166
- installation 166

security

- external user database, using 78
- users 78

security entities, Plone

- about 79
- Global Zope user accounts 80
- permissions 80
- roles 79
- user self registration 80

Sharing tab

- accessing 90
- local roles inheritance 90

Show 'Short Name' on content attribute 57

simple publication workflow

- multiple workflows, adding to single content type 109

- pending state 99
- private state 99
- published state 99
- scripts 108
- state diagram 99
- states tab 104-106
- Transitions tab 106
- Transitions tab, properties 107
- variables 108
- worklist tab 108

site description attribute 57

site title attribute 57

skins.xml file 241

slug, Zope 73

small deployments

- buildouts 256

- layer diagram 256

- monolithic Zope 255

standard Plone content types

- collection 49

- event 49

- file 49

- folder 49

- image 49

- link 49

- news item 49

- page 49

start [process] command 264

status command 264

stop [process] command 264

syndication

- about 199

- folder syndication, enabling 205

- RSS feed, accessing 206

T

table of contents

- about 141

- enabling 141

tail [process] command 264

TAL 95, 215

TALES 95

Taxonomy 9

Template Attribute Language. *See* TAL

**Template Attribute Language Expression
Syntax.** *See* TALES

theming

- best practices 241

- Generic Setup, using 241

- third party add-on products, using 228

third party add-on products

- creating 131

- CSSManager 230

- custom Plone themes 233

- GloWorm 228

third party content types, best practices

- add new content type menu, ordering 146

- content type, superseding 147, 149

- intranet usability, maintaining 149

- rules 145

- upgrades 149

TinyMCE 71

U

unified installer, Plone

- Linux 25

- Mac OS X 27

- Windows 24, 25

users

- administering, ZMI used 87

- managing 82, 84

- password, recovering 86

- password, setting 83

- registration form 82

V

Varnish

- about 268, 270

- building 269, 270

- configuring 269

versioning

- about 199, 207, 208

- Change note field 208

- policy, modifying 208

viewlet

- about 222

- customizing 240

- list 222

- location 223

- managing 224

viewlets.xml file 241

virtual hosting
about 252
domain subdirectory 254
from Apache web server, to Zope
server 253
VirtualHostMonster 252

vs.event
about 154
features 155
installation 154, 155

W

W3C's 8
WCM 7
WCMS. *See* WCM
Web CMS. *See* WCM
Web-based Distributed Authoring and
Versioning. *See* WebDAV
Web Content Management System. *See*
WCM
WebDAV
about 199, 209
access permissions, managing 211
Mac OS X, instructions 209
site structure, viewing as filesystem 210, 211
weblog. *See* blog
Web or Portal Management Systems . *See*
WMS
wget 20
Windows
about 24
external edition, installing 212
WMS
about 8
EMS, differentiating 10
feature 9
work
out-of-box workflows 98
workflow
best practices 114
collective.wtf 93
contents 103
entities 94
features 93
tools 109
workflowstates 12

workflowtransitions 12
workflow, best practices
debugging tools, using 114
initial blueprint, making 114
production servers deployment,
avoiding 114
testing 114
workflow entities
guards 95
local roles, assigning to groups 96
permissions 96
scripts 96
states 94
transitions 94
workflow, intranet
authenticated users access, restricting 123
choosing 123
example, building 124
workflow, tools
collective.workflowed 111-113
collective.wtf 110, 111
placeful workflow 113
Products.DCWorkflowGraph 109
workflow policy 113
workgroup areas
creating 131
World Wide Web Consortium. *See* W3C's

X

XML files, GenericSetup
actions.xml 187
catalog.xml 187
contentrules.xml 187
controlpanel.xml 187
cssregistry.xml, jsregistry.xml 187
factorytool.xml 187
mailhost.xml 187
memberdata_properties.xml 187
portal_languages.xml 187
portlets.xml 187
propertiestool.xml 187
properties.xml 187
rolemap.xml 187
skins.xml 187
types 187
types.xml 187

viewlets.xml 187
workflows 187
workflows.xml 187

Z

ZCA 183

zc.buildout

about 28
directory structure 31
distribute 28
eggs 28
PasteScript 29
running 30, 31
script command modifiers 31
setup tools 28
URL 27
 Zope, launching 36-38
 ZopeSkel 29, 30

ZCML 183

ZEO 257

ZEO, medium deployments

clients 258
clients, adding to buildout 258, 259
clients, performance 258
clients, scalability 258
server, adding to buildout 257

ZMI

about 65, 66, 87
accessing, URL 37
control panel 66

groups, administering 88
portal_workflow 97
URL 23, 66
users, administering 87
view, of Plone site 68-71
workflow management 96, 98

ZODB

about 8, 13
features 14

Zope

several ZODB, attaching to 67
user folder 78
version 19
virtual hosting 252

Zope Component Architecture. *See* ZCA Zope Configuration Markup Language.

See ZCML

Zope Content Management Framework.

See CMF

Zope Enterprise Objects. *See* ZEO

Zope Management Interface. *See* ZMI

Zope Object Database. *See* ZODB

Zope page templates

about 221
METAL 222
TAL 221
TAL, tags 222

Zope server

accessing, URL 37

ZopeSkel 29, 30

Zope Toolkit 183



Thank you for buying Plone 3 Intranets

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

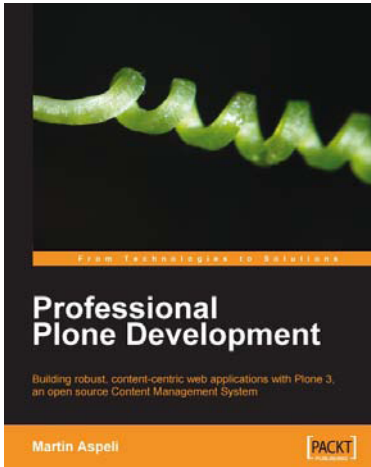
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

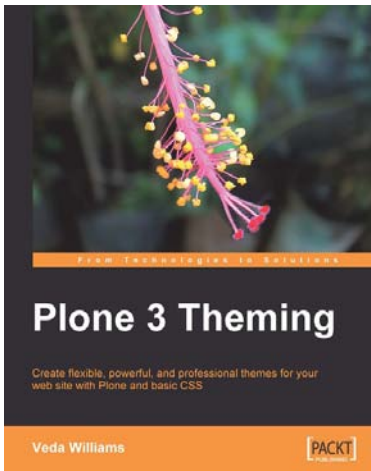


Professional Plone Development

ISBN: 978-1-847191-98-4 Paperback: 420 pages

Building robust, content-centric web applications with Plone 3, an open source Content Management System

1. Plone development fundamentals
2. Customizing Plone
3. Developing new functionality
4. Real-world deployments



Plone 3 Theming

ISBN: 978-1-847193-87-2 Paperback: 324 pages

Create flexible, powerful, and professional themes for your web site with Plone and basic CSS

1. Best practices for creating a flexible and powerful Plone themes
2. Build new templates and refactor existing ones by using Plone's templating system, Zope Page Templates (ZPT) system, Template Attribute Language (TAL) tricks and tips for skinning your Plone site
3. Create a fully functional theme to ensure proper understanding of all the concepts