



F r o m T e c h n o l o g i e s t o S o l u t i o n s

Learning eZ publish 3

Building Content Management Solutions

Leaders of the eZ publish community guide you through this complex and powerful PHP-based Content Management System.

Paul Borgermans
Tony Wood
Paul Forsyth

Martin Bauer
Björn Dieding
Ben Pirt

[PACKT]
PUBLISHING

Learning eZ publish 3: Building Content Management Solutions

Paul Borgermans

Tony Wood

Paul Forsyth

Martin Bauer

Björn Dieding

Ben Pirt

Bruce Morrison



Learning eZ publish 3: Building Content Management Solutions

Copyright © 2004 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 express or implied. Neither the authors, Packt Publishing, nor its dealers or 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 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 edition: May 2004

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

ISBN 1-904811-01-9

www.packtpub.com

Cover Design by www.visionwt.com

Credits

Authors

Paul Borgermans
Tony Wood
Paul Forsyth
Martin Bauer
Björn Dieding
Ben Pirt

Additional Material

Bruce Morrison

Technical Reviewers

Jan Borsodi
Bård Farstad
Balazs Halasy

Commissioning Editor

Douglas Paterson

Technical Editors

Ashutosh Pande
Deepa Aswani

Indexer

Ashutosh Pande

Proofreader

Chris Smith

Layout

Ashutosh Pande

Cover Designer

Helen Wood

About the Authors

Paul Borgermans holds a Masters degree in Science and a PhD in Applied Physics. He became involved with high-performance computing from the start of the 90s while doing plasma physics research. From then on, computing has always been an important part of his job, which gradually involved the integration of large databases, computations, and operational management in R&D. The birth of the World Wide Web also marked the start of projects around intranet developments. Recently, he started a knowledge-management program with his current employer, the Belgian Nuclear Research Center. In this program, eZ publish is used as a cornerstone for developing support tools to capture and preserve knowledge in the long term.

I wish to thank my daughter and wife for the many hours and weekends of patience while working on this book during my free time.

Tony Wood has over 16 years of experience working for both small and large clients, with work ranging from system administration to knowledge management.

Tony is an active member of the eZ publish community and regularly contributes to it. Tony works for VisionWT—a company he founded on the principle that content management should be available to all—which was the first company outside of eZ publish to deploy an eZ publish version 3 site and has focused on creating only eZ publish sites. Tony concentrates on building VisionWT, whatever its budget.

Paul Forsyth is an active contributor to the eZ publish community, and lead developer at Vision with Technology. As Lead Developer, Paul helps to develop processes, methods, and technical standards in addition to developing systems and overseeing the development team.

Previously, Paul was a Software Engineer at Parc Technologies, where his work included programming and system architecture on sophisticated applications optimizing flight schedules and managing fleet size for the airline industry. Paul was a PhD student at the University of Leeds after completing an MSc with Human Computer Systems at De Montfort University and a BSc (Hons) with Artificial Intelligence and Computer Science at the University of Edinburgh.

Martin Bauer is the Managing Director of designIT (www.designit.com.au). Martin draws on a varied background having studied a wide range of subjects including computer science, mathematics, law, literature, and visual arts before graduating from law. Deciding against a career in law, Martin started working as a writer in an advertising agency. This gave him the chance to work on his first website and has been his passion from that point on.

Soon after, Martin joined Creative Access, a reputable web design firm that was purchased in 1999 by Sausage Software and became the head office of Australia's largest web development company, Sausage Interactive. Martin became responsible for a 20+ web development team.

During 2001, Martin worked as a Project Manager for Nebulon, a software consulting firm lead by world-renowned Project Manager Jeff Deluca, creator of one of the most popular agile methodologies, Feature-Driven Development.

In late 2001, Martin joined forces with the original directors of Creative Access to form designIT, which has grown to once again become one of Melbourne's most reputable web design and development firms.

I'd like to thank everyone at designIT for their support and positive attitude without which we could not produce the results we do. I'd also like to thank everyone at the Centre for Design at RMIT for providing us with such a great opportunity. And thanks to my family and friends for always believing in me.

Bruce Morrison is Chief Technical Architect at designIT. Bruce's experience with the Internet started while at Queensland University in the early 1990s. He can recall when all Australian websites could be visited in around 15 minutes, images opened in a separate application, and there were no such thing as tables. Haven't things changed!

Bruce began his career as a System Administrator, working in various fields, including medical research, internet service providers, consultancies, and non-government organizations. In 2002, he turned his back on the heady world of system administration and joined designIT as a Systems Architect.

Bruce has been working with the eZ Publish CMS since the release of the first beta version of 3.0 in December 2002. He has since completed a number of sites using eZ publish.

I would like to thank the great team at designIT for being a terrific bunch of people to work with and my family for their love and support.

Ben Pirt studied architecture at Edinburgh University. In 1999 he moved to New York, where he developed animation skills through working for a design and visualization

company. Here he was also able to strengthen his web development skills through the use of PHP, SQL, CSS, and XHTML. On his return to Edinburgh in 2001, he co-taught a course on architecture and multimedia before completing his postgraduate diploma at the Bartlett School of Architecture. His work with the Interactive Architecture Workshop at the Bartlett enabled him to increase his programming capabilities and to combine this knowledge with other media through embedded programming and hardware design.

Since finishing his diploma, he has continued to develop skills in this area and has extended his web development knowledge through the use of eZ publish in combination with XHTML and CSS. He has recently set up a company, More Associates, where he is able to combine his two main fascinations: design and technology.

Björn Dieding attended the Ev. Gymnasium Werther, where he took advantage of a student exchange to the U.S. in 1997, and at this time, first came into contact with web design. From then on he invested much of his free time gathering knowledge and learning about Internet-related topics. In 1999 he started realizing his first commercial projects. In 2000 Björn attended the b.i.b. Hannover (<http://www.bib.de>), graduating in 2003 as a computer scientist in e-commerce (Staatl. gepr. Informatiker Electronic Commerce). After graduation, he became a freelance web application programmer and web designer. In March 2004, Björn became the first core developer of eZ publish outside eZ systems.

Björn Dieding met Sören Meyer while studying, and at the end of 2002 they decided to team up as xrow GbR (<http://www.xrow.de>). Their main objective was to deliver value-added services for professional and high-quality open source software. Today, xrow GbR is one of the top German consultancies for delivering eZ publish services.

I would like to thank Sören for being my trusted business partner and eZ Systems for their wonderful work on eZ publish.

Table of Contents

Introduction	1
Chapter 1: Installing eZ publish	5
Setting Up	5
Unpacking the Installation	5
Initializing the Database	6
Apache Virtual Host Settings	7
Image Settings	8
ImageMagick	8
GD	8
Cron Jobs	9
Configuration Files	9
The Setup Wizard	10
Page 1 of the Setup Wizard: Welcome to eZ publish	11
Page 2 of the Setup Wizard: System Check	12
Page 3 of the Setup Wizard: Email Settings	13
Page 4 of the Setup Wizard: Choose a Database	14
Page 5 of the Setup Wizard: Database Initialization	15
Page 6 of the Setup Wizard: Language Support	16
Page 7 of the Setup Wizard: Site Packages	17
Page 8 of the Setup Wizard: Site Access Configuration	18
Page 9 of the Setup Wizard: Site Details	19
Page 10 of the Setup Wizard: Site Security	20
Page 11 of the Setup Wizard: Site Registration	21
Page 12 of the Setup Wizard: Finished	22
Troubleshooting	22
PHP Memory Limits	22
PHP Running as a CGI	23
Summary	23

Chapter 2: Content Management with eZ publish	25
What Is Content Management?	25
eZ publish Fundamentals	26
Structuring Content	26
Site Structure	26
Node Tree (Content Object Tree)	26
Sections	27
Content Classes	28
Content Class Attributes	28
Content Object	28
Displaying Content	30
Separation between Content and Presentation	30
Site Structure versus Page Layout and Content Views	30
Overall Page Layout and Content Views	30
Authorization and Roles	31
Disabling a Module/Function outside the Role System	36
Adding Content with the Default Admin Interface	36
Creating Content Classes	37
Datatypes	38
XML Tags Available with ezxmltext Datatypes	46
Datatypes as Information Collectors	48
Creating a New Content Object	48
Editing Objects and Versioning	50
Managing Translations	51
Related Content	53
Workflows	54
Triggers	54
Workflow Events	55
Permissions/Roles	57
Templates	59
More Administrator Functions	61
Removing and Restoring Objects	61
URL Translation	61
URL Management	62
RSS Export and Import	62
Cache Administration	64
Search Stats	65
System Information	65
Section Setup	66
PDF Export	67
Rapid Application Development (RAD)	68

Extension Setup	68
Packages	68
Notification	68
Personal	68
Shop	69
Creating an Example Site	69
Creation of Basic Classes	69
Documents	70
Images	70
Discussion Forums	71
Calendar of Activities	71
Personalization	72
Miscellaneous	72
Creating Classes	72
Taxonomy or Structure	72
Users and Roles	74
Sections Setup	74
User Groups	75
Roles and Role Assignments	76
Summary	78
Chapter 3: Displaying Content with eZ publish Templates	79
Principles	79
Page Layout and Content Views	80
Page Layout	81
Content Views	81
Attribute Templates	82
Template Modularization	82
Style Sheets and Images	82
Edit Templates	82
Templates and Caching	83
Cascading and Overriding Templates	83
Working with eZ Publish Templates	83
Overview	83
Where Does the Content (Data) Come From?	84
Comments	84
Variables	85
Setting and Modifying Variables	85
Variable Types	87
Type Creators	87
Sections in Templates and their Effects on Variables	88

Table of Contents

Variable Namespaces	88
Predefined Variables	88
Using Variables across Templates	93
Controlling Template Output Flow	94
Section	94
If-then-else Constructs with Section	97
Loops with Section	97
Switch Constructs	98
Variable Namespaces Revisited	99
Using Functions from Kernel Modules	101
Functions in the Content Module	101
Fetching a Single Node or Object	104
Fetching Node Lists and Node Trees	105
Counting the Objects of Certain (or all Types)	107
Displaying Version Information	108
Fetching the Current User	108
Others	109
Increasing Performance with Caching	109
Overall Caching	109
Cache-blocks	109
Custom Template Operators	111
The Template Override System	111
Using Cascading Effects in Templates	112
Overriding Templates Using Specific Conditions	112
Syntax of override.ini.append.php	112
Common Template Tasks	114
Navigation Menus	114
Top Level Menu	114
Breadcrumb Navigation	115
Tree Menus	115
Adding Edit Functions to Your Templates	116
Allow Users to Add Content to Your Site	116
Adding an Edit Link	117
Adding a Remove Button	117
Adding a Comment Button	117
Date and Time Tasks	117
Displaying Tomorrow's Date	118
String and Text manipulation	118
Limiting Text Output	118
Limiting XML Text Output	118
Automatic Linking and Conversion	119
Providing a Custom User Experience	119

Creating Dummy Nodes	119
A Specific User Panel	119
Putting the User Preferences Function to Work	120
Showing a User's Groups and Roles	120
Miscellaneous	121
Show a Version History Audit Trail	121
Show Creator, Modifier, and Publishing Date	122
Listing keywords and their Automatically Related Objects	122
Advanced Keyword Facility	122
Creating a Threaded Forum Template	124
Summary	124
Chapter 4: A Glimpse Inside the Core	125
Permissions	125
Object Persistence	128
Getting Attribute Values	130
Setting Attribute Values	130
Other Attribute Functions	131
Persistent Storage	131
Fetching Data	131
Storing Data	131
Other Data Manipulation	132
Content Classes	132
Content Class Attributes	134
Content Objects	135
Creating a Content Object	136
Workflows and Triggers	138
Notifications	142
Information Collection	145
Searching	146
Summary	148
Chapter 5: Extending eZ publish	149
Why Create an Extension?	149
Adding an Extension	150
Locating Your Extension	151
Example Directory Extensions	152
Datatype Extension	152
Module Extension	152

Workflow Eventtype Extension	152
Documentation on Extensions	153
Modules	153
Module Definitions	154
Module Names and Views	154
View Permissions	155
View Parameters	155
View Actions and Post Variables	156
View Navigation	158
Module Coding	159
Reading Module Input	159
Returning Information	159
Processing a Template	160
Redirecting a Module	161
Module Functions	162
Datatypes	164
Datatype Settings	164
Datatype Templates	164
The Datatype Wizard	165
Implementing the Datatype	167
Constructing a Datatype	167
Storing Datatype Information	167
Initializing with Default Values	168
Working with Class Attributes	169
Working with Object Attributes	171
Other Datatype Functions	171
Template Design	172
Complex Datatypes	173
Template Operators	173
Adding a PHP Command	173
The Template Operator Wizard	174
Writing an Operator	176
Registering the Operator	176
Coding the Operator	177
Initializing the Operator	177
Executing the Operator	177
Workflow Events and Triggers	178
Workflow Settings	178
Workflow Events	179
Workflow Triggers	181

Defining Triggers	182
Actions	183
Translations	184
Overriding Translations	185
Notifications	185
Notification Events	185
Adding Collaborations	187
SOAP Server	187
RSS (Really Simple Syndication)	193
Data Interoperability	196
Importing Information	197
Publishing a Folder Object	197
Login Handlers	199
LDAP (Lightweight Directory Access Protocol)	199
Text File Login	201
Summary	201
Chapter 6: Extension Development	203
Extension Development Practices	203
Designing Your Extension	203
Goals and Targets of the Extension	204
Preparing to Test Your Extension	204
Timescales	204
Anticipate the Learning Curve	204
Software Requirements	204
Development Tools	205
Sharing with the Community	205
Documentation	206
Creating the WorldPay Extension	206
Creating the Environment	207
Creating Workflow Events and Triggers	208
WorldPay Module	212
Creating the Module Extension Environment	213
Creating the Module	213
Reviewing the ini Settings	217
User Settings	217
Permissions	218
Callback Testing	218
Creating the Category Datatype	219

Category Datatype Design	219
Setting Up the Extension Environment	219
The Category Database Table	220
Database Communication	221
Category Discussion	221
Category Templates	224
The Category Datatype in Action	224
Integrating Existing Code with eZ publish	225
Making a Bridge to External Applications	225
Strategies	226
Who Am I?	227
Authentication	227
Communicating with Google	230
Modifying Existing Code	233
Summary	233
<u>Chapter 7: Deploying eZ publish</u>	<u>235</u>
Define Your Hosting Requirements	235
Number of Visitors	236
Security Needs	237
Reporting Requirements	238
Budget	239
Time Limits	239
Shared or Dedicated	239
Is My Server Powerful Enough?	240
Documentation	241
How and When to Update the Documentation	241
Contact Details	241
Location	242
Hardware	242
Operating System	242
Software	242
Patching Process	242
DNS Information	242
TCP/IP Information	242
Access Control	243
Upgrade Roadmap	243

Disaster Recovery	243
Preparing the Linux Environment	243
Apache	244
PHP	245
Database (MySQL/PostgreSQL)	246
GD Graphics library	247
ImageMagick	247
Cron Jobs	247
SMTP	248
PHP Accelerators	248
Deploying	249
New Project Deployment	250
Updating Project Deployment	250
Backups	251
Ports	251
Summary	252
Chapter 8: Center for Design at RMIT Case Study	253
The Client	253
The Existing Site	255
The Project	255
The Process	256
Requirements	257
Key Objectives	257
Creative	257
Functionality	257
Content	258
Hosting Environment	258
Selecting a CMS	258
Specifications	259
User View	259
Admin View	261
Links	263
Miscellaneous	264
Content Model	265

Display Templates	267
Sustainable Products, Sustainable Buildings, and LCA Template	268
Content Types	269
Interface Design	272
Visual Design	272
HTML Prototype	274
The Home Page	275
Section Pages	275
Content Pages	277
Development	278
Install eZ publish	278
Define Content Classes and Sections	279
Configure Roles and Permissions	279
Apply Display Logic and Templates	280
Create Page Layout Templates	281
Navigation	282
Setting Up	283
Summary Pages	292
Content Templates	296
Testing	300
Requirements	300
Specifications	300
Implementation	300
Functional Testing	300
Content Population	301
Deployment	301
Maintenance and Support	301
Training	301
Project Assessment	302
Requirements and Specification Phases	303
Development Phase	303
Content Population and Review Phase	304
Extending the Site	304
Workflow	305
Archiving	305
Integration with CRM	305
Summary	305

Chapter 9: Creating a Standards-Compliant eZ publish Site	307
What Are Web Standards?	307
XHTML	308
CSS	309
Web Standards: Real-World Scenario	310
Accessibility	310
Bandwidth	310
Future Proofing	310
Ease of Maintenance	310
eZ publish and Web Standards	311
The Client Requirements	311
Planning and Preparation	314
Template Design	316
page_head.tpl	319
header.tpl	320
navigation.tpl	321
image.tpl	322
footer.tpl	323
CSS Rules	325
Designing the Content	328
The News Article Class	329
The Data Class Definition	329
Class Templates	329
CSS Rules	330
Performance	332
View Caching	333
Template Compiling	333
Template Cache Blocks	333
PHP Accelerators	334
Benchmarking	335
ab	335
Effects of Optimization	336
Summary	337

Appendix A: Template Operators and Functions	339
Operators	339
String Operators	339
String Transformations	341
Counting and Comparing Strings	344
Array Operators	344
sum and sub	348
inc and dec	348
div	348
mod	349
mul	349
Max and min	349
abs	349
ceil and floor	349
round	349
Localization and Translation Operators	349
Logical Operators	351
ne	351
lt	351
gt	351
le	351
ge	351
eq	351
null	352
not	352
true	352
false	352
or	352
and	352
choose	353
contains	353
Type Checking	353
Image Handling	354
Other Template Operators	357
count	357
Accessing Variables in the ini Files	358
cond	358
first_set	359
eZ publish Kernel Operators	359
Index	363

Introduction

eZ publish is an open-source Content Management System created by eZ systems of Norway (www.ez.no). Built primarily upon around the LAMP platform—Linux, Apache, MySQL and PHP—eZ publish is based on open standards, with a flexible content model, a powerful template engine, workflow management, role-based permissions, and much more. Now at version 3, eZ publish has matured beyond a Web-based Content Management System to a powerful, Content Management Framework, which developers can extend and customize to produce unique web solutions.

There is also a set of commercial licenses for eZ publish, but the free version places no restrictions on your usage of the system.

Like most open-source products, eZ publish has a large and dedicated community whose members share their experience and knowledge with the rest of community. At:

<http://www.ez.no/community>

you will find forums, contributions, bug reports, documentation, news, and more.

eZ publish is powerful and versatile, and as such, many have found it difficult to get to grips with initially, with much hair-pulling over some of its seemingly esoteric features. This book owes much to such hair-pulling and cursing—our authors have come through that experience, and the material we present here will help you develop the skills required to become an accomplished eZ publish developer.

What This Book Covers

The structure of this book is as follows: we begin with installing eZ publish, look at how to implement and customize a site, and then extend and deploy the system. The book is rounded off by two real-world case studies of eZ publish implementations, and the design choices and strategies that were employed to realize them.

Chapter 1 starts us off with a straightforward guide to installing an eZ publish instance. Once we are setup, *Chapter 2* moves on to look at the fundamental features of eZ publish—its content handling and the structure of an eZ publish site. The chapter takes us through the eZ publish administration area, and shows how to use it to work with content, users, and permissions.

Chapter 3 takes us into eZ publish's template system—through templates, we control our site's output, and this chapter shows the general principles of the template engine, an in-depth discussion of the template language and syntax, and the template-override

system. The chapter is linked to *Appendix A*, which discusses the various operators and functions. Before we embark on a detailed exploration of eZ publish extensions, *Chapter 4* looks at how some of eZ publish's fundamental concepts are realized at the code level.

eZ publish is designed to be extended, and in *Chapter 5* we look at the different types of extension that can be created for eZ publish. Extensions allow new template operators, datatypes, and modules (among others) to be developed separately and added to the base eZ publish system. An extension may be something as small as a new template operator, or something as critical as an interface to your other business applications. In *Chapter 5*, we tackle how to use the extension mechanism to add *your* functionality to eZ publish.

In *Chapter 6*, we look at real-world examples of extension development, create new modules and datatypes, and explore strategies for integrating existing code or systems with eZ publish.

Chapter 7 covers strategies and techniques for successfully deploying eZ publish projects, from assessing and preparing your hosting environment to moving the project to your production server.

The final two chapters of the book are real-world case studies of eZ publish 3 implementations. In the first case study, we see the journey from client requirements to content model, and from HTML mockup to template design. eZ publish requires you to think about the structure of your site and the types of content it supports in a way that may be different from how you usually view your site—this first case study provides a vivid illustration of how to approach this problem. The second case study shows us how eZ publish, XHTML, and CSS fit together to produce a standards-compliant eZ publish site, with particular emphasis on the template design.

What You Need for Using This Book

To use this book, you will of course need eZ publish. This is freely downloadable from http://www.ez.no/ez_publish/download.

If you have working installations of PHP, Apache, and MySQL, you can download the source code for eZ publish. Alternatively, there are installer packages that install all the required software to run eZ publish (PHP, Apache, and MySQL). Note that these packages are meant for testing purposes only.

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.

There are three styles for code. Code words in text are shown as follows: "For the category class, the `category_string` attribute returns a comma-delimited string listing the categories the attribute belongs to".

If we have a block of code, it will be set as follows:

```
<?php
include_once( "kernel/classes/ezdatatype.php" );
define( "EZ_DATATYPESTRING_NEWDATATYPE", "newdatatype" );
class newDataTypeType extends ezDataType
{
    function newDataTypeType()
    {
        $this->ezDataType( EZ_DATATYPESTRING_NEWDATATYPE, "None" );
    }
}
ezDataType::register(EZ_DATATYPESTRING_NEWDATATYPE, "newdatatype")
);
?>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines will be made bold:

```
<?php
include_once( "kernel/classes/ezdatatype.php" );
define( "EZ_DATATYPESTRING_NEWDATATYPE", "newdatatype" );
class newDataTypeType extends ezDataType
{
    function newDataTypeType()
    {
        $this->ezDataType( EZ_DATATYPESTRING_NEWDATATYPE, "None" );
    }
}
ezDataType::register(EZ_DATATYPESTRING_NEWDATATYPE, "newdatatype")
);
?>
```

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen—in menus or dialog boxes, for example—appear in our text as follows: "clicking the Next button moves you to the next screen".

Tips, suggestions, or important notes appear in a box like this.

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

```
mysql> create table books (name char(100), author char(50));
Query OK, 0 rows affected (0.03 sec)
```

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 drop an e-mail to feedback@packtpub.com, making sure to mention the book title in 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.

Downloading the Example Code for the Book

Visit <http://www.packtpub.com/support>, and select this book from the list of titles to download any example code or extra resources for this book. The files available for download will then be displayed.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or a code error—we would be grateful if you could report this to us. By doing this you can save other readers from frustration, and also help to improve subsequent versions of this book.

If you find any errata, report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the Submit Errata link, and entering the details of your errata. Once your errata have been verified, your submission will be accepted and the errata added to the list of existing errata. The existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Questions

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

1

Installing eZ publish

Installing eZ publish can seem like quite a daunting task at first, but if you follow the guidelines it's one of the easiest content management systems to set up. In this chapter you will learn how to install and configure eZ publish once you have set up the server.

There are two parts to this chapter: the first part deals with where to place the files and how to initialize the database, and the second part takes you through the setup wizard.

Setting Up

There are several ways of performing the steps in this guide, which depend upon whether you have command line access to the server or not. In each example, instructions on using either of these setups are provided. I have focused on UNIX-based installations because these are the most common environments in hosting. Windows users can still follow these steps, but will need to find Windows equivalents of the commands here.

Unpacking the Installation

The first thing to do is download the latest distribution of eZ publish from <http://ez.no/>. The archives are normally available in `tar.gz`, `tar.bz`, and `.zip` file formats. Windows users should choose the `.zip` format for compatibility, and *NIX users can choose from the `tar.gx` or `tar.bz` formats. The most compatible method is to expand this archive on your own computer and then use FTP to transfer the files to the correct folder on your server.

Since the archive is only about 4MB in size compared to the 20MB size of the expanded archive, it makes sense to upload the archive and then extract it rather than uploading the extracted files individually. If you have command line access to the server, you can use the following UNIX commands to do this:

```
wget http://ez.no/downloads/ezpublish.3.x-x.tar.gz
tar -xzf ezpublish-3.x-x.tar.gz
```

All files needed for installation will be extracted to a folder called `ezpublish-3.x-x` (where `x` is the version number). These files should then be transferred to the correct folder on your web server. Some web hosting packages have powerful control panels (such as CPanel or Ensim) that allow you to expand archives through their interface. If this is the case, you can upload the archive and use the control panel to expand it.

Once you have expanded the archive, you will find a number of files and folders. The most important of these are shown in the following table:

Name	Function
<code>design</code>	This is where all of the default eZ publish templates are stored and is where you will create your new ones.
<code>index.php</code>	This is the main controller script for eZ publish. All requests to the website come via this.
<code>kernel</code>	This folder contains the main scripts that make eZ publish function.
<code>lib</code>	This folder contains all the additional libraries that add functionality to eZ publish.
<code>runcronjobs.php</code>	This script should be run regularly to ensure that various tasks are performed.
<code>settings</code>	This folder contains all of the default settings, as well as all the override settings that you will create.
<code>var</code>	This folder is used to store all the images and files that get uploaded and provides a location to store all the cache files that are generated.

Initializing the Database

MySQL is the most commonly used database for eZ publish, so we have used it in these examples. All you need to do at this stage is create a database for eZ publish to use, and assign a user for that database. On many shared hosting solutions, a database already exists for you, and all you need to do is note down its name and the user name and password used to access it (which should have been provided with the hosting package). In case you need to set up the database yourself, perform the following steps:

1. With command line access, log in to the server on which the database is hosted and use the following command to launch the MySQL application:
`mysql -u username -p`
2. This will then prompt you for your password. Once you have logged in to the database, issue the following command to create the database:
`CREATE DATABASE database_name;`

This will create the database, but it will be accessible only to the user that you logged in as. For security reasons it is better to create a new user for this database, because if someone else were to somehow read your configuration files, your password would be compromised, because it is stored in plain text.

3. To create a user who only has access to this database, issue the following command:

```
GRANT ALL PRIVILEGES ON database_name.*
    TO username@localhost
    IDENTIFIED BY 'password';
```

If you do not have access to the command line, the best method is to install a web-based database administration tool such as phpMyAdmin, which will enable you to create the database and user using a web browser.

Apache Virtual Host Settings

Although it is possible to run eZ publish in a subfolder of a website, it works extremely well when set up properly as a virtualhost in Apache with `mod_rewrite` enabled (this also increases security, because all requests are directed to `index.php`). eZ publish uses URLs such as `/node/view/253` to view a node in the CMS. `Mod_rewrite` means that this URL could be translated into something like `/contact/`, which is far more meaningful and memorable to the end user. Most hosting services have `mod_rewrite` enabled as it is on by default in Apache. If you can't access your `httpd.conf` file for apache, it is still possible to enable rewriting by using a `.htaccess` file in the root of your eZ publish directory.

If you have access to the settings for your virtual host, you need to set it up as follows:

```
<VirtualHost *>
    DocumentRoot /path/to/ezpublish
    ServerName test.ez.local
    <Directory /path/to/ezpublish>
        Options FollowSymLinks Indexes ExecCGI
        AllowOverride None
    </Directory>
    RewriteEngine On
    RewriteRule !\.(js|gif|css|jpg|png)$ /path/to/ezpublish/index.php
</VirtualHost>
```

The most important lines in this code are the rewrite statements at the end. If you already have a working virtualhost configuration, just add these two lines to it, rather than copying this whole setup. If you are able to override the settings for `mod_rewrite` in your `htaccess` files (check with your hosting provider), you can place a file called `.htaccess` containing the following lines in the root of your installation. This is the default name, and what 99% of hosting providers use, but it pays to check first!

```
RewriteEngine On
RewriteRule !\.(js|gif|css|jpg|png)$ /path/to/ezpublish/index.php
```

Image Settings

eZ publish uses server-side image manipulation utilities to create the different versions of images that you will upload. For example, if you upload a large photograph, eZ publish can resize it to a size you specify and then create thumbnails at a size you specify. There are two main utilities that do this: **ImageMagick** and **GD**. ImageMagick is the more powerful of the two, being able to apply a number of effects (such as border, blur, twirl, etc.) to the image, but both perform equally well for the simple requirement of resizing.

ImageMagick

ImageMagick is the more powerful of the two utilities, and is used to add many different effects to the images you upload. It is a standalone binary located on the server and often needs to be installed manually. If your server is running on a UNIX-based system and you have full access to the machine, installation is simple. Download the binary for your system (or the source if you have an unusual setup) from <http://www.imagemagick.org> and follow the guide for installing it. If you are running Windows, it is also possible to download a binary for your system from this source. If you have limited access to your system, first check if your hosting provider is willing to install it. Otherwise, depending on your system limitations it may be possible to upload the binary. If you do perform a custom installation, remember that eZ publish must be able to locate the convert binary before it can use it. If stored in a normal location, it will be found automatically, but if it is in a custom location, specify this in `/settings/override/image.ini.append.php`:

```
[ImageMagick]
IsEnabled=true
ExecutablePath=/path/to/binary
Executable=convert
```

GD

GD is a library for image manipulation that has to be compiled into PHP to run. In recent versions of PHP (>4.3), GD is included by default, which makes life a lot easier. GD is less powerful than ImageMagick, and is normally used only as a fallback if ImageMagick is not found. Hosting providers running the latest versions of PHP will obviously have GD, but those running older versions of PHP are unlikely to add it to their existing installation as it requires rebuilding PHP. Visit <http://www.boutell.com/gd> for more information on the GD library.

If you are using a GD version prior to v2, image scaling will look bad because those versions used a very simple re-sampling algorithm.

Cron Jobs

It is necessary to have a regularly executed script to execute various tasks that need to be carried out periodically. These tasks include the execution and processing of workflows and the issuing of notifications to users via e-mail. On the most common setup, a UNIX-based system, the best way of doing this is to set up a **cron job** that runs approximately every 15 minutes. A cron job is simply an automated process that uses a script to carry out various tasks periodically. Most systems have **crontabs** for individual users; these are programs that allow users to define cron jobs. You will need to edit the crontab file for your system and add the following lines:

```
#This must be set to the directory where eZ publish is installed.
EZPUBLISHROOT=/path/to/your/ezpublish/directory

# Location of the PHP Command Line Interface binary.
PHP=/usr/local/bin/php

# Executes the runcronjobs.php script every 15th minute.
0,15,30,45 * * * * cd $EZPUBLISHROOT; $PHP -C runcronjobs.php -q 2>&1
```

This will execute the `runcronjobs.php` script from the command line every 15 minutes. If you are not able to set up this task, the site will still be useable, but some features, such as the ones mentioned, will not function properly. On a Windows system you need to use the Scheduled Tasks mechanism to execute a batch file, which would execute the `runcronjobs.php` script.

Configuration Files

eZ Publish is configured through a collection of files known as **ini files**, located in the settings directory. The **Setup wizard**, which we discuss in the next part of this chapter, automatically defines some settings for these files, but it is necessary to understand how these are used in order to modify an eZ publish installation.

In the root of the settings folder are the default configuration files that come with the installation. Two folders, `override` and `siteaccess`, are used to modify these default settings to make the site work for you. The `siteaccess` folder is used to configure multiple views of the site and contains subfolders that correspond to the name set up for the site. For example, there may be subfolders called `admin` and `plain`. The order in which these locations are read is:

1. The default configuration files in the settings folder
2. The files in the siteaccess folder corresponding to the current view
3. The files in the override folder

The order of precedence of files according to their location is
`override > siteaccess > settings`.

Although there are many configuration files (about 30 at the last count, and more are added every time a new feature is added) you generally need to tweak only a few:

Configuration File	Comments
site.ini	The main configuration file for the site; covers database settings, site access list, and cache settings
override.ini	Defines the template overrides for the different sites defined
image.ini	Defines the different image presets for the sites
i18n.ini	Defines the different language setups for the sites

In general, there will be a `site.ini` file in the `override` folder, which defines the database access for the entire site because the database access will be the same for both the user site and the admin site. More site-specific settings, such as the template overrides in `override.ini`, will be in the `siteaccess` folder because they vary between the different site accesses. The files `image.ini` and `i18n.ini` will often be specific to individual sites because these settings depend upon the audience and the design.

The eZ publish configuration files can have a number of different file endings: `.ini`, `.ini.append`, and `.ini.append.php`. eZ publish will search for files with all of these endings in the relevant folders.

The recommended extension is `.ini.append.php`, because if the file were somehow requested through the web server, it would be served as a standard PHP file, and since the configuration lines are interpreted as standard PHP comments, there would be no output.

Several ini files will be discussed throughout the book as we encounter more features of eZ publish that need to be configured and controlled.

The Setup Wizard

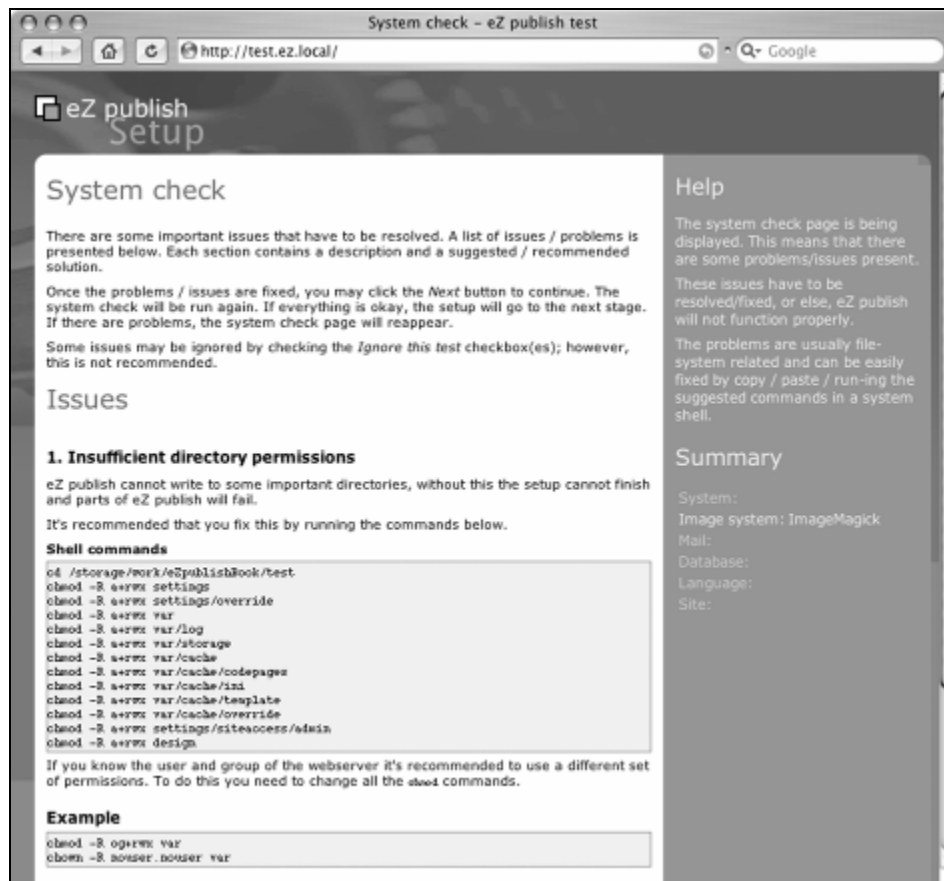
eZ publish has a very user-friendly setup system that takes you through all of the parameters necessary for setup and checks the system to make sure that everything is configured correctly. We will now go through all of the steps in turn to explain what they mean and what needs doing at each stage. Point your browser to the location in which you extracted the eZ publish setup files to start the **Setup Wizard**.

Page 1 of the Setup Wizard: Welcome to eZ publish



This is the first page of the setup wizard. By clicking Next you will be taken through a series of steps to configure eZ publish for your system. It will automatically detect the image system you are able to use. If you have ImageMagick installed it will use that, otherwise it will attempt to use GD if it is compiled into PHP. You can tell if it has found a usable system by looking at the bottom right part of the next screen, under Image System.

Page 2 of the Setup Wizard: System Check



The setup wizard checks the permissions for various directories that eZ publish uses to run. These are:

/var	Used by eZ publish to store all the image files and the cache files
/settings	Used by eZ publish to store the settings
/design	Used by eZ publish to store the design templates

If the permissions are not correct, you can use the script shown on the page to set their permissions, or use your FTP client to change the permissions for each folder to 777, which means they are writeable by the web server. There is an issue with this command—if you are on a shared hosting environment, other users may be able to access these files. If this is a problem, you should look for dedicated hosting. Note that if you are

able to change the owner of files on the web server, you can make these files belong to the web server user and disallow access to others. The command to do this is:

```
chmod -R og+rwX var
chown -R user.group var
```

where `user.group` is the username and the group that the web server process runs under.

If everything is already set up properly, you will not see this page. When you click Next, the check will be run again unless you have selected Ignore this test.

Page 3 of the Setup Wizard: Email Settings

The screenshot shows a web browser window titled "Email settings - eZ publish test" with the address bar showing "http://test.ez.local/". The page is titled "Outgoing E-mail" and contains the following content:

This section is used to configure how eZ publish delivers its outgoing E-mail.

There are two options:

- Direct delivery through sendmail (must be available on the server).
- Indirect delivery using an SMTP relay server.

E-mail delivery:

☒ sendmail

☐ SMTP

Server name:

Username (optional):

Password (optional):

< Back Next >

Help

The eZ publish system uses E-mail to send out important notices such as user registration and content approval. On Linux/UNIX: try to use sendmail. On Windows: use an SMTP server.

Sendmail:

Mail is delivered directly using the sendmail transfer agent. The sendmail binary is usually available on most Linux/UNIX systems. If sendmail is not available then SMTP should be used.

SMTP:

Mail is delivered through an SMTP server. At the minimum, the hostname of the SMTP server must be specified. Hint: check the SMTP settings in your E-mail application.

Summary

System: OK
Image system: ImageMagick
Mail:
Database:
Language:
Site:

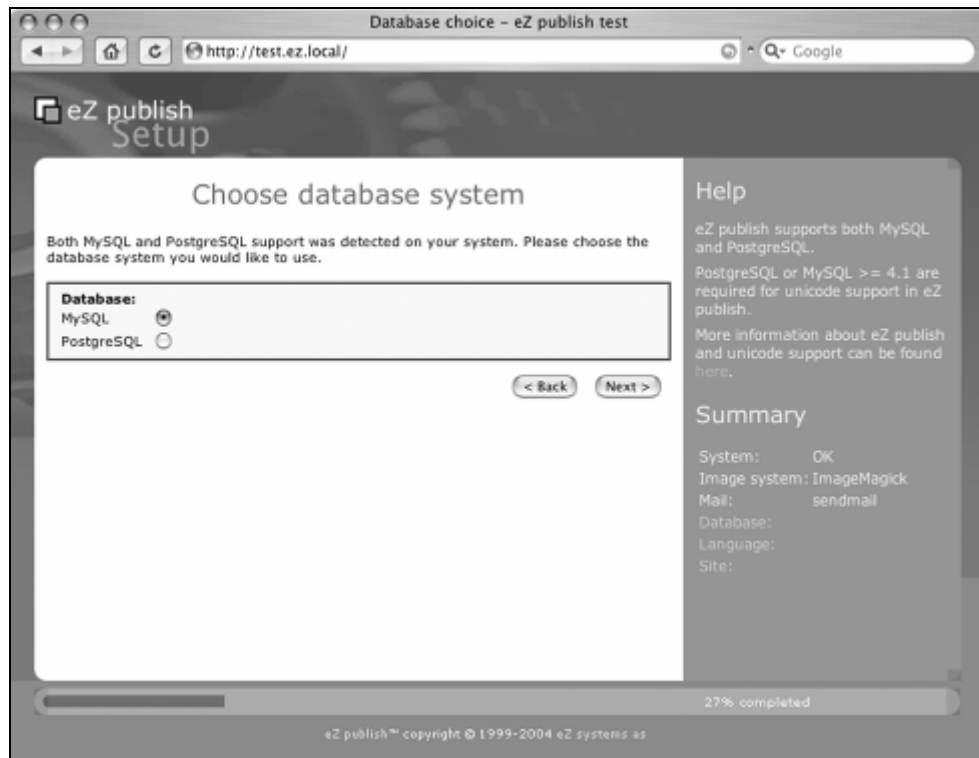
18% completed

eZ publish™ copyright © 1999-2004 eZ systems as

For eZ publish to send out e-mails to users, it must be configured to use a mail server. If you are using a UNIX-based system, you can normally use the local **sendmail** program.

If you are using Windows or for some reasons do not have sendmail installed, you can use an external SMTP server to send your mails.

Page 4 of the Setup Wizard: Choose a Database



This is a particularly simple stage and lets you choose between MySQL and PostgreSQL as your database of choice. PostgreSQL includes some more advanced features that are not present in MySQL, although these do not greatly affect the running of eZ publish.

If you don't have a reason for wanting to run PostgreSQL, you probably don't need to!

Page 5 of the Setup Wizard: Database Initialization

The screenshot shows a web browser window titled "Database initialization - eZ publish test" with the address bar showing "http://test.ez.local/". The page features the "eZ publish Setup" logo. The main heading is "Database initialization" with a sub-instruction: "Please input database access information in the form below." A form titled "Database" contains the following fields: "Type" (set to "MySQL"), "Servername:" (set to "localhost"), "Username:" (set to "username"), "Password:" (set to "password"), and "Socket (optional):" (empty). Navigation buttons "< Back" and "Next >" are at the bottom of the form. To the right, a "Help" section explains that eZ publish requires a database and provides instructions for MySQL and PostgreSQL. Below the help is a "Summary" section listing: System: OK, Image system: ImageMagick, Mail: sendmail, Database: MySQL, Language: (empty), and Site: (empty). A progress bar at the bottom indicates "36% completed" and the footer shows "eZ publish™ copyright © 1999-2004 eZ systems as".

When we set up the database user earlier in this chapter, we defined a username and password. This is where you enter them. You do not need to select a database at this point because eZ publish will only let you choose from the databases you have access to.

If eZ publish cannot connect to the database using the credentials you have supplied, you will have to repeat this step until it can. Recheck the user name, password, and privileges you set up earlier if this happens.

Page 6 of the Setup Wizard: Language Support



eZ publish has very strong support for multi-language installations. At this stage you can choose a default language as well as any other languages you need setting up on the system. Languages can be added at a later stage if necessary, so do not add languages here if you aren't sure you need to.

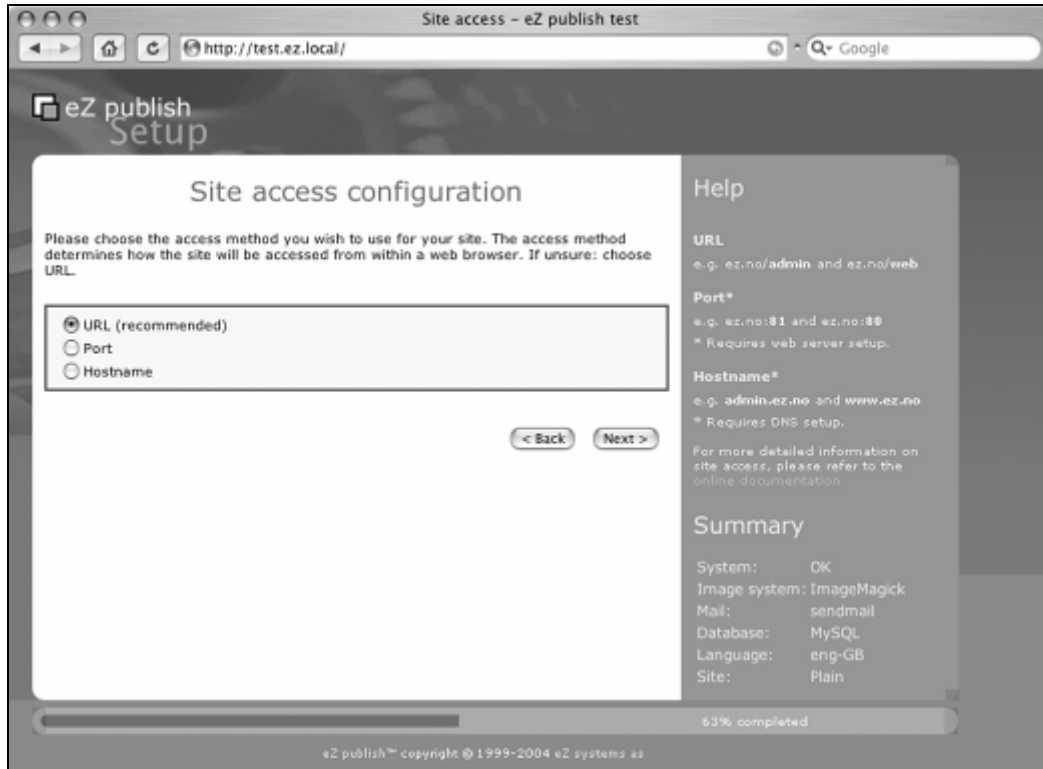
Page 7 of the Setup Wizard: Site Packages



The best way of getting to know eZ publish is by installing one of the pre-configured site packages and playing around with it. By default, eight site packages come bundled with eZ publish to fulfill a number of different needs. Choose the Plain site package if you are planning on building your own site from scratch. Otherwise, it may be simpler to modify an existing package. Each site is unique and requires its own database.

Site	Description
Blog	A simple blog setup with dated posts
Corporate	A multi-section site with a simple corporate look and feel
Forum	A user forum, similar to the forums at http://ez.no
Gallery	A powerful gallery using eZ publish's image-handling system
Intranet	A typical corporate intranet
News	A news site with news stories in different categories
Plain	The no-frills setup to use if you need to roll your own site
Webshop	A sample e-commerce application using eZ publish's shop functionality

Page 8 of the Setup Wizard: Site Access Configuration



The different sections of the site can be accessed in a number of different ways.

Access Type	Description	Example URL
URL	The simplest method of accessing different sections of the site.	<code>http://www.mysite.com/admin</code>
Port	It is possible to assign a port through which to access your site. This is more complex to set up and requires custom Apache configuration.	<code>http://www.mysite.com:81/</code>
Hostname	Specific host names can be used to decide which part of the site is accessed. This is more complex to set up and requires custom Apache configuration as well as adding DNS entries for the different hosts.	<code>http://admin.mysite.com</code>

Page 9 of the Setup Wizard: Site Details

Site details – eZ publish test

http://test.ez.local/

eZ publish Setup

Site details

This page lets you modify information about the sites you've chosen to install. In addition, it also lets you choose a database for each site.

Information about the sites you've chosen to install is shown below.

Site details

Title: Plain

Site url: http://test.ez.local/

Admin e-mail: admin@localhost

User path: plain

Admin path: plain_admin

Database: test

< Back Refresh Next >

72% completed

eZ publish™ copyright © 1999-2004 eZ systems as

At this stage you can choose the title for the site and set up the URL that will be used to access it. It's also possible to fill in the administrator's e-mail and set up the paths that will be used to access the admin and user sections of the site. If the database already contains some data, this page will reappear and ask you what to do. Possible actions are:

- Leave the data and add new
- Remove existing data
- Leave the data and do nothing
- I've chosen a new database

Page 10 of the Setup Wizard: Site Security



If you cannot use the virtualhost mode, you will need to use an **htaccess** file to secure your system. There is an example .htaccess file in the root of the distribution that if renamed from .htaccess_root to .htaccess will make the setup secure. If you are running in virtualhost mode you will not see this screen.

Page 11 of the Setup Wizard: Site Registration

Registration - eZ publish test

http://test.ez.local/

eZ publish Setup

Site registration

If you wish, you can register this installation by sending some information to eZ systems. No confidential data will be transmitted and eZ systems will not use or sell your details for unsolicited emails.

The registration email:

Comments:

Site info:

Template: - Plain

Title: - Plain

If you wish, you can also add some comments, which will be included in the registration E-mail.

Comments:

Thank you for a great CMS :-)

Note: Sending out the email and generating your site might take a couple of seconds. Please wait until the next page loads. Clicking the button again will only send out duplicate emails, and may corrupt the installation.

☒ Send registration

< Back Next >

Help

By sending registration the following data will be sent to eZ systems

- System details (OS type, etc)
- The test results
- The database type
- The site name
- The url of the site
- Languages chosen

This data will help to improve future releases of eZ publish.

Summary

System: OK

Image system: ImageMagick

Mail: sendmail

Database: MySQL

Language: eng-GB

Site: Plain

90% completed

eZ publish™ copyright © 1999-2004 eZ systems as

This page of the setup allows you to send an e-mail to eZ systems to let them know the details of the set up of the system you are using. Although this is optional, it is good practice because they will be able to focus their developments upon the areas that are useful for the most users.

Page 12 of the Setup Wizard: Finished



At this stage the site setup is finished. Make a note of the URLs on this page as they will be used to access your site in future.

Troubleshooting

Once you have reached this point you should have a fully functional eZ publish site to start working with. If you've hit a snag, there are a few common issues you can look at to get things working.

PHP Memory Limits

A common problem in hosting environments is that the memory limit of PHP is set too low. The default value of the limit is 8MB, but this is too low for eZ publish to run (although the eZ team has been working hard at reducing memory consumption). If you get output similar to the following, then you have this problem:

```
Fatal error: Allowed memory size of 8388608 bytes exhausted (tried to
allocate 184320 bytes) in /www/ezsite/kernel/classes/ezpackage.php on
line 2140
Fatal error: eZ publish did not finish its request
```

The execution of eZ publish was abruptly ended, the debug output is present below.

There is an easy fix to this if you are allowed to override the PHP settings using a .htaccess file (if your host doesn't support this, you may need to look for a new host) placed in the root of the eZ publish installation. In this file, add the following line:

```
php_value memory_limit "16M"
```

This will raise the memory limit to 16MB. If this still does not work, continue raising the limit until it does. In general, you will not have to go over 32MB for the site to work.

You may need to increase this setting later, if your pages are very complex.

PHP Running as a CGI

If your hosting provider is running PHP as a CGI instead of an Apache module, then eZ publish will have problems. The main symptom of this is that regardless of the link you click on, you will always get the front page of the site. The only way of fixing this problem (in versions of eZ publish prior to 3.4) is to modify `ezsys.php`, one of the eZ publish files. There are a number of posts in the forums at <http://ez.no> describing how to do this.

Fortunately, one of the outlined features for version 3.4 of eZ publish is a fix for this problem, so the best thing is to use this version when it becomes available.

To tell if your host is running PHP as a CGI, create a PHP file on your server containing the following script:

```
<?php
if (php_sapi_name() == 'cgi')
    echo "PHP is running as a cgi";
else
    echo "PHP is running as a module";
?>
```

Summary

In this chapter we've discussed the steps for a basic eZ publish installation. We've looked at initializing the database, the Apache web server settings, setting up the image manipulation libraries, and scheduled jobs, and discussed the role of the ini file in eZ publish configuration.

We also walked through the Setup Wizard, which greatly eases the installation and configuration of eZ publish, and looked at some common installation problems.

Now that your system is installed and configured, it is time to move on. In the next chapter, you will start using eZ publish as a tool for content management.

2

Content Management with eZ publish

The prime objective of eZ publish is web-based content management. With the powerful, flexible content object model, it is easy to map any kind of structured information into a collection of attributes. The basic architecture of eZ publish includes version management, multi-language possibilities, relations between pieces of information, and all kinds of interactions and manipulations defined and offered by standard or custom-built modules.

This chapter will guide you through the basic concepts and paradigms of eZ publish. At the end of the chapter, we will create an example site to show how the concepts and architecture map to a concrete eZ publish site—this is something that you will see more of in the two case studies later in the book. By then, you should have a sound idea of what eZ publish is all about.

What Is Content Management?

Content management covers the processes and workflows involved in organizing, categorizing, and structuring information resources so that they can be stored, published, and reused in multiple ways.

A Content Management System (CMS) is used for collecting, managing, and publishing content.

Content is stored either as components (including metadata) or as whole documents (in the form of binary file attachments), while maintaining the links between components or documents. The CMS preferably also provides revision control by keeping track of versions. Workflow is usually a part of the game in the form of electronic implementations, where for example, approvals and notifications are performed through e-mail and/or online in the CMS itself.

eZ publish fulfills the needs and roles of a web-based CMS where interaction with the client (browser) is over standard HTTP and related protocols (including WebDAV and SOAP). It is therefore perfectly possible to make links (through some additional programming) with traditional desktop environments, even though the internal modular design and architecture is agnostic towards your preferred desktop environment.

With respect to documents, eZ publish supports its own basic XML format for structuring content, which includes basic elements such as headers, lists, tables, and some embedded or linked media types like images and movies.

eZ publish Fundamentals

The first step is to create a design or architecture for your content. This is done before you start with the implementation, but the powerful and sometimes unique features of eZ publish may influence your design decisions. You are encouraged to explore this chapter, the tutorial, and the two case studies in this book, and experiment with eZ publish before undertaking your first real project.

Structuring Content

We will begin with the site structure, followed by a drilldown through concepts such as node trees, sections, object classes, attributes, and features or functionality that are important for managing eZ publish-based websites through its modular kernel system.

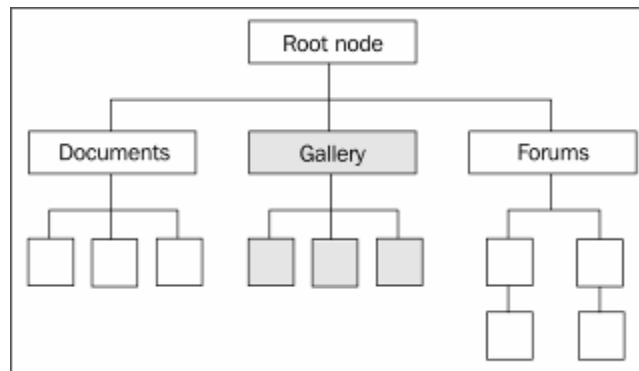
Site Structure

How to structure your site depends a lot on the types of content you want to serve, the audience, and how you want clients to interact with the site content. There are no fixed rules on how you should design the structure of a site, only some good practices that you can borrow from people who create taxonomies for document collections.

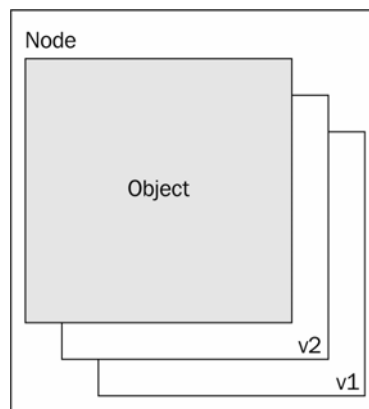
If you are on tight time schedules and already have some experience with eZ publish or another CMS, you can start immediately by implementing a global outline (through the admin interface) and site design with the base eZ publish distribution. However, the design and implementation of major projects requires a good deal of blackboard brainstorming and careful global planning.

Node Tree (Content Object Tree)

Once you have determined the global structure, you will find that a direct reflection of such a structure is given by a node tree. The following diagram gives an example of a simple structure implemented in an eZ publish node tree:



It includes a document structure, ordered galleries, and a forum branch. The node tree is a structured way of placing content objects (the real information). This distinction in eZ publish is important because content objects can have more than one placement (node).



A single node is in fact the placement of a collection of object versions and translations sharing the same object ID (a number in the eZ publish database implementation). The object name is shared by versions of the same translation.

Sections

Sections are virtual collections of nodes that belong together, either conceptually or functionally. They are used as a mechanism in role-based permission management and are also design parameters. Once a node is assigned to a certain section, newly created child nodes inherit this section information by default. There is always at least one section for content objects that is assigned by default.

In the role-based permission system, sections can be used to control access and the type of actions users or user groups can perform. In their bare essence, sections categorize

your content to enable an additional layer of presentation logic; for example, you could use a special navigation panel for each section or even change the entire page layout.

Content Classes

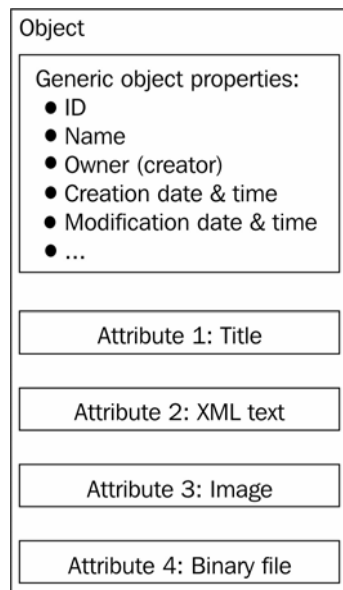
Content classes define the way content is structured for a certain content object type. This flexible system—where any number of base datatypes (called content class attributes) can be used for structuring information into workable units of information—is one of the prime features of eZ publish.

Content Class Attributes

Content class attributes are the lowest level of information 'typing', although some of them are already powerful compound datatypes. eZ publish contains a rich set of 30 datatypes. A few more can be found on <http://ez.no> as community contributions.

Content Object

Content objects are simply instances of certain classes. The content elements are therefore specified by the class attributes. The following diagram shows an example object after creation. Besides the attributes defined in the corresponding class, there are a number of generic properties such as an object ID, name, and creation date among others. The name of a content object is usually bound to one or more of the content object attributes, such as a title.



Content Object Version

An object in eZ publish is always under version control. The versioning can be controlled to maintain only a certain number of versions, possibly only one.

Versions are handled by the parameters in the settings file `content.ini`, in the `[VersionManagement]` section, where you can specify the maximum number of versions to keep. You can either specify a global default with `DefaultVersionHistoryLimit` or specify versions per class with an array containing the class ID as the index and the number of versions to keep as a value. For example, `VersionHistoryClass[8]=2` will allow the minimum of versions (two) for the class with an index number of 8.

You cannot specify `VersionHistoryClass[8]=1` because then you will not be able to edit the object after creation. This is because a temporary version is created as a draft version while editing an object.

Older versions of objects are kept within the limits of the version history settings. This means objects older than the current 'published' version are set into an 'archive' state. When there is more than one user editing objects, it could so occur that there are still a few draft versions lingering around in the underlying database tables between the archived versions.

eZ publish allows you to revert to an older version of an object by copying the older version to a new draft and publishing it again. The history of objects is always preserved within the limits of version history as described earlier. This can be of importance if you should use eZ publish as a document management system, where quality systems impose the maintenance of document versions.

In the current state of eZ publish, spotting the real differences between object versions is not easy unless a UNIX `diff`-like tool is used. However, the construction of an audit trail listing the object history is pretty straightforward.

Content Object Attributes

These are instances of content class attributes. It is important to note that the version management applies to all attributes, including binary files that may be stored in the eZ publish system.

Content Object Translations

Objects may be translated into languages other than the default-configured *main language*. If for some reason, you do not want translations for certain attributes, they can be disabled at the attribute level. A typical example is a photograph, which is usually language independent.

Displaying Content

In this chapter, content has been treated in an abstract way. However, in most practical situations, the content should reach and appeal your target audience in some way or another. As eZ publish is mainly a web-based CMS, you will want to format the content in a clear and attractive way for display inside a web browser. eZ publish enables you to do so by way of the built in template engine that transforms your content in a manner of your choice for delivery; by default, it transforms it as XHTML output.

Separation between Content and Presentation

One of the fundamental architectural properties of eZ publish is the separation between content, application logic (to manipulate content), and the content presentation for either a browser or other output media such as PDF files (introduced in eZ publish 3.3).

Site Structure versus Page Layout and Content Views

In principle, the structure of your site (node trees) is independent of the page layout and content views (introduced in the next section). However, by carefully planning your sites in the design stage, you may come up with a solid model that can equally be used for parts of the page layout. You can gain more insight into this process from the example site we will create at the end of this chapter, and also from the two case studies later in the book.

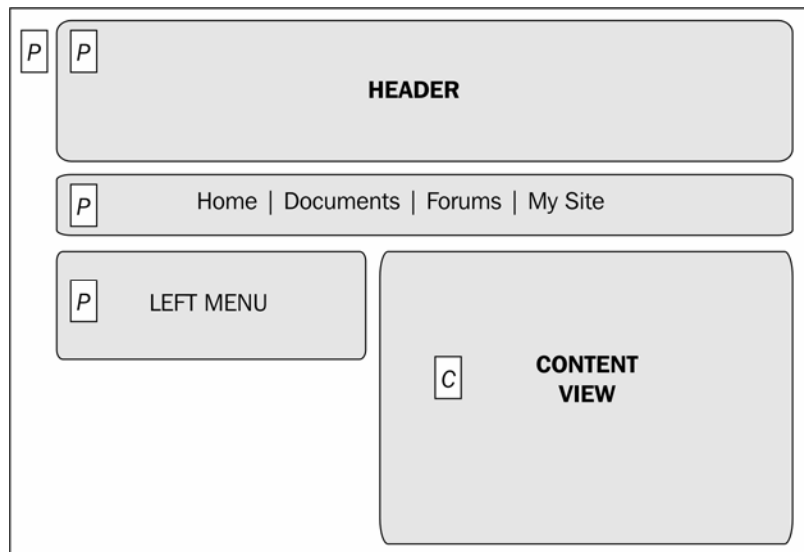
One such example is the navigation and menu structures used. You can, for instance, use the top nodes as top menu or navigation entries and second-level nodes (most of the time these will be folder types) as entries for further navigation.

Overall Page Layout and Content Views

In the paradigm that eZ publish uses, there are two main 'areas' where content is delivered: the overall **pagelayout** and **content views**. Navigational elements are mainly bound to the page layout templates, and the main content views are normally placed inside the global page layout. You can assume for now that the content views are extracted from the database and kernel modules depending on the URL you use to browse an eZ publish-based site.

Examples of content views are the display of articles and other complex content, folder listings, and forms, but also the content from managing functions such as template manipulation, role definitions, and the output from custom-built extensions.

The following figure gives an overview of this simple concept. The parts marked P are generated in the page layout template(s), while the part marked C is generated with content view templates.



The concepts and a detailed explanation of the template system are covered in Chapter 3.

All the items mentioned earlier are accessible in the same object-oriented paradigm inside the templates.

Authorization and Roles

The eZ publish permission system is based on the concept of roles. This is different from what you may be used to in a traditional file system where permissions are set on individual files and directories (with inheritance). The latter is still possible with the eZ publish role-based system as it provides granularity down to the level of individual nodes or a node and all of its children. In general, setting permissions or roles for individual nodes should be avoided and applied only to special cases as this may complicate the administration of a site.

The role-based system goes further; you also can specify the functions of kernel modules that can be used by a given role. For example, you can specify that a normal user can add comments to articles, but not submit articles themselves.

A role in eZ publish consists of one or more **policies** with details of modules, functions, and limitations that apply to the particular role. For any module or function, you can omit limitations altogether; for example, in the case of the default administrator user who has access to all modules and functions. An example is the content module. This module handles most of the functions dealing with the management of content. The most obvious function is `view()`, which fetches the content objects for sending to the browser through

the template system. Limitations for this function can specify one or more content classes, a section, any special conditions (access specifications to modify content), a single node, or a set of one or more node trees (subtrees).

The following table gives an overview of the kernel modules and functions that are available for role assignments in eZ publish version 3.3. The access-control limitations listed are to be used in the policies for a role.

Module	Function	Access-Control Limitations	Comments / Definition
content	read	Class, section, owner, node, subtree	Read access. A finer specification can be made for "self" or "any".
content	create	Class, section, parent class, node, subtree	A create policy also needs an edit policy.
content	edit	Class, section, owner, node, subtree	Allow a user to edit content objects. A finer specification can be made for "self" or "any".
content	remove	Class, section, owner, node, subtree	Allow a user to remove objects. A finer specification can be made for "self" or "any".
content	bookmark	None	Allow a user to make bookmarks that are stored inside eZ publish.
content	translate	Class, section, owner, node, subtree	Allows a user to add translations of an object.
content	versionread	Class, section, owner, status, node, subtree	The status can be draft, published, pending (workflow), rejected (workflow), or archived (older versions).
content	pendinglist	None	Shows the pending objects; these are objects that are the subject of some workflow.
content	urltranslator	None	Provides access to the urltranslator.

Module	Function	Access-Control Limitations	Comments / Definition
content	cleantrash	None	Allows the clearing of the trash can.
content	translations	None	Access to the definition of languages available (to be used only in the default admin interface).
class	None	None	Access to the class edit functions.
collaboration	None	None	Access to the collaboration module.
error	None	None	Access to the error module.
ezinfo	None	None	Access to the eZ publish info page with version, PHP configuration, and so on.
form	None	None	Access to the form module (simple form processing, which is different from information collection).
layout	None	None	Access to the layout module for overriding the current page layout (for example, a print layout).
notification	use	None	Access to the notification mechanism (adding/deleting notifications).
notification	administrate	None	Running the notification subsystem outside the cron jobs.
package	read	Type	Package type can be class, site, site style, and patch. Only classes and site style are implemented in version 3.3.

Module	Function	Access-Control Limitations	Comments / Definition
package	list	Type	See package/read.
package	create	Type, creator type, role	Type: see package/read. Creator type can be content class export or site style. Role is not a role in the role system but acts as a classification of the package: lead, designer, developer, tester, or contributor.
package	edit	Type	See package/read.
package	remove	Type	See package/read.
package	install	Type	See package/read.
package	import	Type	See package/read.
package	export	Type	See package/read.
pdf	create	None	Access to create PDF generation definitions.
pdf	edit	None	Access to edit PDF definitions: edit.
reference	None	None	Access to the doxygen generated documentation.
role	None	None	Access to the roles for editing. Currently assigned roles can be read in the user object inside templates.
rss	feed	None	Access to the RSS feed or import definitions.
search	None	None	Access to the search module.

Module	Function	Access-Control Limitations	Comments / Definition
section	None	None	Access to section definitions and edit functions.
setup	None	None	Access to the setup functions.
shop	administrate	None	Access to the shop administration.
shop	buy	None	Access to buy items.
shop	setup	None	Access to the shop setup functions (discount rules, VAT schemes).
soapserver	None	None	Access to soapservers (currently no soapserver services are defined by default).
trigger	None	None	Access to the trigger/workflow administration.
url	None	None	Access to URL administration.
user	login	siteaccess	Specifies who can login to which siteaccess.
user	selfedit	None	Access to users for editing their own data.
user	password	None	Permission to users to change their passwords.
user	preferences	None	Access to user-bound preferences (variables).
workflow	None	None	Access to workflow administration.

Currently (in eZ publish 3.3), the role system is limited to the level of objects/nodes as the smallest unit. You cannot apply roles at the attribute level (yet). If you really need to, you can implement permission-related features inside templates. For example, you could

check the (main) user group of the current user and display part of the attributes in view and edit templates.

Disabling a Module/Function outside the Role System

In certain situations, you may want to disable a module or function altogether, and not just for specific roles. This can be done in the `site.ini` configuration file. For example, you could choose to turn off the `user/register` function so that new users can only be defined by an administrator:

```
[SiteAccessRules]
Rules[]=Access;disable
Rules[]=Module;user/register
```

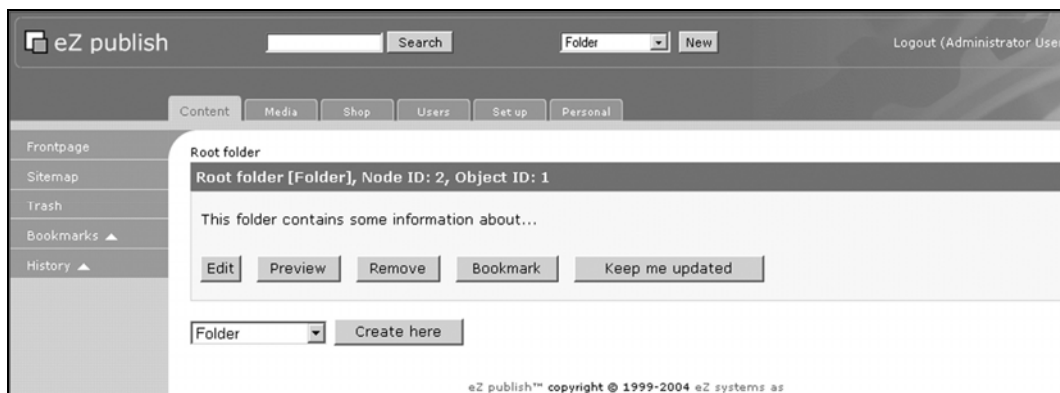
After disabling the `user/register` function, you will also need to edit the template for user login to avoid the function being called, otherwise this could result in an error message that is not very user friendly.

For an intranet site, you will probably not need the `shop` module or any of its functions:

```
[SiteAccessRules]
Rules[]=Access;disable
Rules[]=Module;shop
```

Adding Content with the Default Admin Interface

In this section, we will walk through the administration site access. The functionality offered by the administration site access can be used to understand the functionalities of the major underlying kernel modules.



When you first log in to the admin site, you are directed to the content section. At the top, you will notice the various areas of the admin site. The top menu tabs give access to the Content, Media, Shop, Users, Set up, and Personal sections.

Creating Content Classes

To create a new content class, go to the Set up menu and choose Classes from the advanced part in the left menu box.

The first screen gives an overview of the class groups. Class groups are used to group similar classes by either content type or functionality. For example, the user's class group holds the basic user-related classes for both individual users and user groups.

When entering a certain class group (or after creating a new group), you have the option to create a new class. Any new class has a few generic properties:

- Class name
- Class identifier
- A name pattern for objects created as instances of this class

The latter is important, as it will determine the URL part of this object when URL translation is configured.

With the new button and selection, you can add new attributes based on one of the 30 standard datatypes available. In the following screenshot, the generic attributes are shown as well as the first attribute created for the new class: a title that is defined as a text line datatype (also called ezstring). For the Object name pattern, the title is inserted as the primary component for subsequent object names.

If you change the object name pattern later, objects already published will not adhere to the new name pattern unless they are re-published.

The screenshot shows the 'eZ publish' administration interface. The top navigation bar includes 'Content', 'Media', 'Shop', 'Users', 'Set up', and 'Personal'. The left sidebar lists various system components, with 'Advanced' expanded to show 'Classes'. The main content area is titled 'Editing class - My new class' and shows the class details. The class is named 'My new class' with identifier 'my_class' and object name pattern '<title>'. It is associated with the 'eZ publish book' group. Below this, a message states 'Input was stored successfully'. The 'Attributes' section shows a single attribute named 'Title' (Text line) with identifier 'title'. The attribute is marked as 'Searchable' and has a 'Max string length' of 0. The interface also includes a 'Member of groups' table and a 'Required' checkbox.

Name	Identifier	Default value	Max string length	Required	Searchable	Information collector	Disable translation
Title	title		0	☐	☒	☐	☐

Datatypes

This section lists the available datatypes along with possible applications. The table with indicators for 'Searchable' and 'Information Collector' is meant to indicate whether the datatype can be indexed in the search engine or be used as an information collector.

ezauthor

The ezauthor datatype can be used to store one or more authors. The subfields are the author name and e-mail address. This datatype also checks for valid input for the e-mail address. This datatype actually can hold more than one author and can be used to specify a list of authors for a certain object (like a document).

Searchable	Information Collector
Yes	No

The ezauthor datatype is not related to the users defined in your eZ publish site.

ezbinaryfile

The binary file datatype can be used to store any file. The indexing for the search engine is done with external programs if the content is not simple ASCII text. The configuration for these external handlers is done in `settings/binaryfile.ini`.

You can specify an upload limit size in MB (or 0 for no limits). You can use this datatype to upload legacy documents, for example Acrobat PDF or MS Word files.

Searchable	Information Collector
Yes	No

ezboolean

This datatype corresponds to a checkbox-type datatype. It is typically used for controlling further template elements, like the display of a button for adding comments.

Searchable	Information Collector
Yes	Yes

ezdate, ezdatetime, and eztime

The date- and time-related datatypes allow for adding date and time as attributes. The input validation code checks for correct values. Although you cannot use these attributes

in the normal search functions, you can use them in custom searches inside templates like most other datatypes (see Chapter 3).

Searchable	Information Collector
No	No

The `ezdate` values are stored as UNIX timestamps in the database. Therefore, you are limited to a certain date range depending on the platform.

The year range on Linux/UNIX is 1901 to 2038. On Windows-based systems, the range of valid years includes only the years 1970 through 2038. If you need a larger range, you need to either use a string function, which of course has its limitations, or create a new extended date datatype in PHP.

Note that the validation of input data is not foolproof; some checks are not performed on date items; for instance, there are no checks on the limits of date items, such as on the day of a month. This could possibly result in unwanted behavior—you could try to enter a non-valid date like March, 56 2004 and the system might not complain.

ezemail

A single e-mail address can be stored with this datatype. The input is validated for the correct format. Currently, this check only applies to the format, and not the actual existence of the e-mail address or its domain part.

Searchable	Information Collector
No	No

ezenum

The `ezenum` is deprecated from version 3.3 onwards because of its complexity and performance issues related to it. For new classes, you should avoid this datatype in favor of the `ezselection` datatype. This datatype is included in the eZ publish package only for reasons of backward compatibility.

With the `ezenum` datatype, you can present the user with a predefined set of values. These can be either specified as a single selection or multiple selections. The style can also be set to either use radio buttons (single selection) or checkboxes (multiple selection). This datatype is similar to the `ezselection` datatype discussed later.

Searchable	Information Collector
Yes	No

ezfloat

The `ezfloat` datatype stores floating-point numbers. You can specify the minimum and maximum value at the class level for input validation.

Searchable	Information Collector
No	No

ezinteger

The `ezinteger` datatype is similar to the `float` datatype where you can specify a minimum, maximum, and default value. Contrary to the `float` datatype, integers can be searched for with the search engine.

Searchable	Information Collector
Yes	No

ezidentifier

The `ezidentifier` datatype can be used for constructing (sequential) identifier strings such as `doc-xxx-en`, where `xxx` is a numerical value with a predefined number of digits. You can set a prefix and a suffix, the number of digits to use, and the number part to start from. The values for the number part are created automatically and cannot be edited. The number part does not increase when new versions of objects are created. You need to be careful when you update the attribute definition; for example, when you change the prefix and/or suffix, you must change the initial value at its primary setting. Identifiers assigned earlier on remain unchanged.

Searchable	Information Collector
Yes	No

You could use this datatype to automatically assign a serial number for articles, photographs, or products.

ezimage

The `ezimage` datatype is used to store bitmaps in various formats. The supported formats are defined in `settings/image.ini`, where you can specify a number of predefined sizes

with a label (such as medium, small, large). Upon uploading the image, new variations in size for the image are created according to the defined sizes.

Searchable	Information Collector
No	No

ezinisetting

This is a special datatype that can be used to modify or display various INI settings. The INI setting to be modified is defined in the class by specifying the *siteaccess*, *ini file*, *ini section*, and *ini parameter*. In addition, the type of setting needs to be specified and can be one of the following: text, enable/disable, true/false, integer, float, and array. To determine the type of setting you need to use, you need to look at the settings in the default INI file(s) shipping with eZ publish.

A word of caution here: since the content of this datatype is determined at the class level, it is possible to have more than one object that can alter an INI setting. On the other hand, this may be wanted behavior for cases where you want different administration profiles. For example, you can create a class that allows for the modification of a large set of INI settings along with a class with the ability to modify a restricted set of INI settings.

Searchable	Information Collector
No	No

ezisbn

The *ezisbn* datatype is used to store an ISBN; the input validation checks for the correct ISBN format.

Searchable	Information Collector
Yes	No

ezkeyword

The *ezkeyword* datatype can be used to specify keywords as a list of comma-separated strings. An advantage of this datatype is that it can be used to link objects of the same class; whenever there is one or more common keyword, a list of keyword-related objects is available for use when displaying the content object. Note that the relations are within the same class. When you use keyword attributes in different classes, there is no automatic linking of related objects.

Searchable	Information Collector
Yes	No

This datatype can be used to enhance the user experience: for example, in a document archive where you can provide a See also list of documents that share one or more keywords with the current object.

ezmatrix

The `ezmatrix` datatype can be used for data that is best stored as a matrix or in a table. You need to specify the maximum number of columns and the initial number of rows. Upon object creation, the author can add more rows to those initially specified.

Searchable	Information Collector
Yes	Yes

This datatype comes in handy when you need to input a varying number of items within a content object. With clever template programming, the matrix datatype becomes one of the most versatile available in the standard distribution of eZ publish.

ezmedia

The `ezmedia` datatype is used for storing media clips and currently supports the following formats: Flash, RealPlayer, Quicktime, and Windows media. You can also specify a maximum file size to upload, or 0 to remove any size limit.

Searchable	Information Collector
No	No

ezobjectrelation

You can use this datatype to specify one related object upon creating or editing an object. This datatype can be useful when you need to assign a single item out of a pool of existing objects to a newly created object. There is no way to restrict the object class of the related object.

Searchable	Information Collector
No	No

ezobjectrelationlist

The object relation list is a powerful datatype that can even be used to create compound objects (of different classes). Alternatively, it can be used to create categories of related objects. When the creation of objects is enabled, you also can specify a restricted set of classes from which the object can be composed. A further parameter specifies the default node at which newly created objects can be placed.

In the `content.ini` file, under the `[ObjectRelationDataTypeSettings]` section, you can specify parameters to assign default nodes to start browsing. The node can be specified by the node name or the node ID.

Searchable	Information Collector
Yes	No

ezoption

This datatype can be used to specify a number of options upon **object creation**. This is most useful for collecting information through forms. At object creation, you specify the options with a name and value pair.

Searchable	Information Collector
No	Yes

ezpackage

The `ezpackage` datatype is used in administration classes to specify the available packages. You need to specify the type as a text field, and the mode of displaying the available packages as a combo box or a group of icons. For this, you can specify an image such as a screenshot for guiding the user with each package.

Searchable	Information Collector
No	No

ezprice

The `ezprice` datatype is used in product classes for representing prices. If one of the attributes is of this type, the standard view templates automatically add buttons for adding the object to a shopping cart or the wish list of a current logged in user.

Searchable	Information Collector
No	No

ezselection

The **ezselection** datatype is used to specify a number of options from which you can specify single or multiple values. The definition of the options is done upon **class creation** and comprises a certain number of values—this is different from the **ezoption** datatype, where the options are for object creation.

Searchable	Information Collector
Yes	Yes

A modified version of this datatype where you can specify an option name and value has contributed by the eZ community.

ezstring

The string datatype is used to hold short strings, such as titles, with less than 255 characters.

Searchable	Information Collector
Yes	No

ezsubtreesubscription

The **ezsubtreesubscription** datatype is used for the notification system and behaves like a boolean variable (checkbox) upon object creation (not class creation). When the creator of an object selects this checkbox, a notification rule is added, where the creator gets e-mail notifications when the object is updated or child nodes are created/changed.

Searchable	Information Collector
No	No

eztext

The **eztext** datatype is used for plain text, such as simple forum messages. The size of the text is limited only by the database and possible browser constraints. By default, eZ publish uses the **text** datatype to store the value of a text field in MySQL (65,534 bytes). You can choose larger sizes by modifying the database setup for versions up to 3.3. From version 3.4 onwards, the default size is increased to MySQL medium text fields.

Searchable	Information Collector
Yes	Yes

ezurl

As its name implies, the `ezurl` datatype is used to store the URL (contents of the `href` attribute) and a display string. The interesting feature with eZ publish is that the link is also stored separately in the database to allow link-checking with a cron job or some other scheduler. When a URL becomes invalid, you can change this to a new value from the admin interface, and all instances of the URL's use are automatically updated.

Searchable	Information Collector
No	No

ezuser

The `ezuser` datatype is a special datatype that is connected to the User module of eZ publish. Indeed, you can create several user classes in your system, grouped under the User class group in the class list. This datatype specifies the following properties:

- A login name
- An e-mail address
- A password

You can use this feature for several types of users, mainly with different metadata (attributes) depending on the type. Although modifying the base user class is in principle not a problem, you can take the safest approach and create your own. To use it as the default User class, you can set the class ID in `site.ini` under the `[UserSettings]` section.

Searchable	Information Collector
Yes	No

ezxmltext

The XML text datatype is used to store rich-text content through a well-structured yet simple variant of XML. It supports some elements of the XHTML 2.0 specification and is further characterized by the possibilities of specifying special input and output handlers. For instance, there are handlers to format the contents in plain XML, XHTML, or PDF (introduced in eZ publish 3.3).

Searchable	Information Collector
Yes	Yes

There is one caveat: your editors or end users will have to learn the syntax for providing or editing this rich-text content.

A browser-based DHTML editor known as the **Online Editor** is available from eZ systems. It offers WYSIWYG editing for the easy creation of structured (and formatted) text.

The structured text is rendered as XHTML, and you have control over the display through the template system.

XML Tags Available with ezxmltext Datatypes

As the `ezxmltext` datatype is used for enabling structured blocks of text, it provides several XML tags, by default, which you can use. Furthermore, it is possible to extend the available XML tags through custom-defined XML entities.

All XML entities can have a `class` parameter. This parameter is normally used in the tag templates to specify a CSS class, but you are free to use sophisticated transformations with the eZ publish template language. For example, a header can be made of the `chapter` class and will be rendered in a more complex layout than available in CSS alone.

Headings

Headings and titles can be tagged by making use of either the `<h>` or the `<header>` tag. You can specify two optional parameters: `level` and `class`. The `level` parameter can be used to define the size or the level of the heading.

```
<header [level="1-6"] [class="packt"]>My header</header>
```

Lists

Lists are tagged in the same way as their XHTML counterparts for ordered and unordered lists with the help of the `<ol><li>...</li></ol>` and `<ul><li>...</li></ul>`.

Lists **cannot** be nested in eZ publish up to version 3.3 (and, at the time of writing, 3.4 as well).

Emphasizing Text

To emphasize text portions, you can use a variety of XML tags that correspond to their use in traditional XHTML markup. By default, emphasized text can be rendered in italic or bold typefaces. Here is a list of available tags:

- `<i [class="test"]>Italic text.</i>`
- `<em [class="test"]>Emphasized text.</em>`
- `<emphasize [class="test"]>Emphasized text.</emphasize>`
- `<b [class="test"]>Bold text.</b>`
- `<bold [class="test"]>Bold text.</bold>`
- `<strong [class="test"]>Bold text.</strong>`

Literal (Unformatted) Text

The `<literal>` tag can be used to render unformatted text, for example, program source code, HTML code, XML source, and so on. Everything that is inside a literal block will be rendered in the same way (character by character) as it was input into the XML text field. By default, this effect is achieved using the XHTML `<pre>` tag.

Hyperlinks

Hyperlinks can be inserted by making use of the `<a>` or `<link>` tags. Both tags accept a target parameter, which is used in the same way as the corresponding XHTML parameter. It accepts the `_self` or `_blank` values. The `href` parameter is required, and needs to be set to a valid URL.

In addition to the `href` parameter, a parameter called `id` can also be used. This is because eZ publish stores every hyperlink used in XML text datatypes in a separate `ezurl` table. The automated link checking (enabled through cron jobs) validates these URLs, and is very convenient when you need to fix any broken links. The `id` parameter should be assigned a value (number) corresponding to the `ezurl` ID.

The `<anchor>` tag makes it possible to insert XHTML anchors inside the XML text datatype field. An inserted anchor will work just like a standard XHTML anchor: it accepts a `name` parameter that should be unique in the collection of anchors used in the final rendered XHTML output.

Tables

Tables in XML text datatype fields are created in the same way as in XHTML. The `<table>` tags accept values for the `border` and `width` parameters. Rows are created with the `<tr>` tag, table headers with the `<th>` tag, and table data with `<td>` tags. The `<th>` and `<td>` tags accept the `width`, `colspan`, and `rowspan` parameters.

Objects

The `<object>` tag is a special and unique eZ publish feature that can be used to embed other eZ publish objects inside XML text fields. This is typically used for embedding

images, but you can use an object of any class you define. It accepts a parameter ID corresponding to the object ID (not the node ID!).

Further parameters can be used to specify alignment (with `align` accepting the values `left`, `right`, or `center`), size (a size parameter that accepts `small`, `medium`, and `large` as values), and a `view` parameter with values `embed` or `text_linked`. Finally, `href` and `target` parameters can be specified to turn the embedded object into a link to whatever you wish.

Custom Tags

You can also define your own custom tags. A custom tag is defined inside `[CustomTagSettings]` in an override for the `content.ini` configuration file. It accepts a `name` parameter that corresponds to the definition in the `content.ini` configuration file. In accordance with CSS stylesheets, you can specify a custom tag as being inline (like `<i>`) or a block style (like `<p>`).

By default, a few custom tags are defined in the standard eZ publish distribution, such as `<superscript>` and `<subscript>`.

Datatypes as Information Collectors

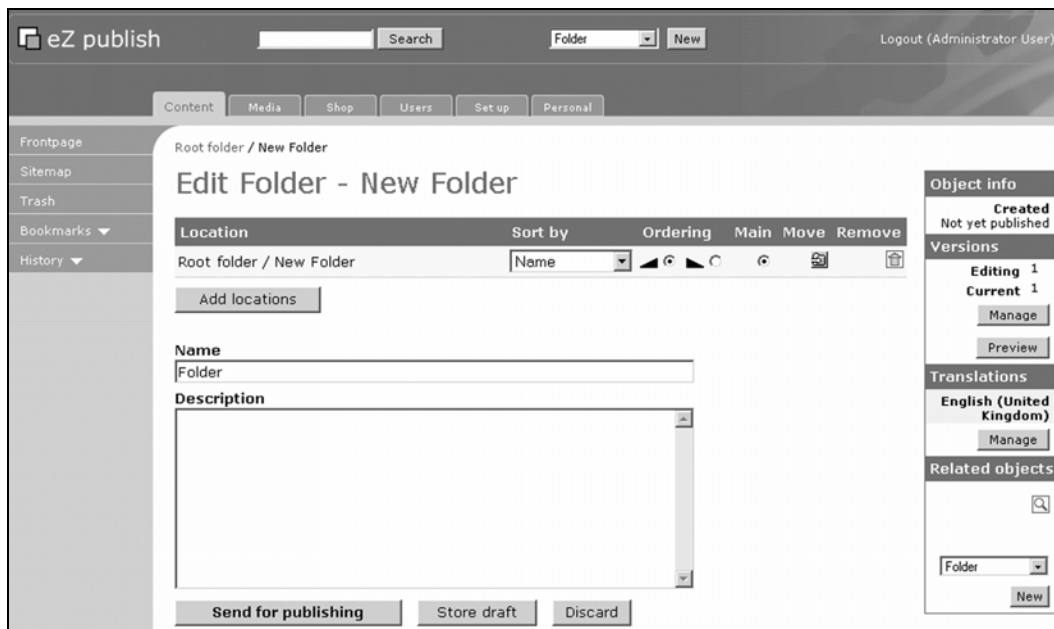
Some datatypes can serve as **information collectors**. Objects created with datatypes marked as information collectors can be used as a mechanism in forms where data is sent to a preconfigured e-mail address or stored in the database (to make polls, for instance).

This can be useful in small surveys or feedback-type applications. The main way to collect information for your site is of course by creating new objects that are placed in an appropriate node in the node tree.

Creating a New Content Object

New content objects are created in the Content section of the admin area. In this section, you can browse through the node tree and create new objects in certain locations.

The following screenshot shows the Content section, where we want to create a new object (a simple folder).



The selection box shows the newly created class, after pressing the Create Here button; a new edit screen is then presented where you can fill in the required and optional attributes for the object created.

A small toolbar is displayed on the edit screen, with information on the current location, the default sorting method for child objects, and the sort order (ascending and descending).



The Main field in the toolbar is used for determining the main node location for an object.

By using the Move button, you can change the main location by browsing through the node tree.

With the Remove icon (the little trashcan icon), you can delete one or more locations assigned for this object.

Clicking on the Add locations button brings up a browse view of the node tree that you can use to select additional locations for the object.

The following screenshot shows the edit screen for a newly created article object. From this point on, there exists a draft object with a corresponding object ID and location specified by the node ID.



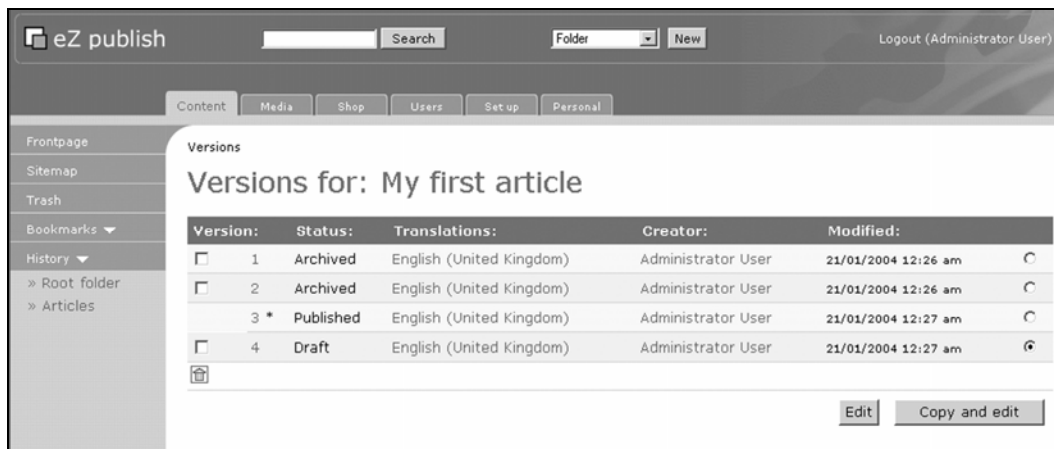
Further in the edit screen, you can create content by filling in the attribute parts.

The prime effect of adding additional node locations is that additional links can be created. For example, you could assign a new location to an article inside the news tree of your site, for display on the default page that a user sees when accessing your site. When a normal (full) view of the object is requested, the display will be done according to the main object placement parameters.

When you add another placement, the child nodes do *not* inherit these extra placements. For applications where you need to display the children at these locations, you will have to resort to template programming, look up the main node placement, and fetch the children on the basis of their IDs.

Editing Objects and Versioning

With the version management part in the edit screen, you can go back and copy one of the older versions to a new draft and publish it again. This is illustrated in the following screenshot, where you can either continue to work on the current draft (provided this draft is created by you) or select an older version for copying to a new version. A radio button is used to select the version you want to edit or copy and edit. The selection button is used to remove versions.



Managing Translations

If you did not specify the available languages for translation during the installation of eZ publish, you can still add them in the Set up menu as shown in the following screenshot:



To add translations for content objects, edit an object and then click the Manage button in the Translations section of the edit screen that you saw earlier. This brings you to a screen where you first select the additional languages (provided that you have configured

your site to be multilingual) after which you can add a translation to your new content object.



By selecting the language you want to translate the content into and clicking the Add button, you arrive at the translation screen. This is shown in the following screenshot, where the original text is displayed on the right side of the screen along with the newly translated version so as to guide the translator.



Related Content

Relating content objects in eZ publish is a very powerful feature that can be accomplished in two ways:

- By the general object relation mechanism available with any content object.
- By using one of the two special datatypes (`objectrelation` or `objectrelationlist`). The object relation can hold a single relation while the object relation list can hold one or more object relations.

The following screenshots show how you can add a generic object relation to an object (by clicking the magnifying glass icon in the Related objects panel in the edit screen). The action is fairly straightforward from there on, as you can browse through the node tree and select an existing object.



Clicking the Select button brings you back to the edit screen where the newly related object is added to the list in the Related objects panel:



The page on which all users should start browsing can be customized (by default it is the main content node) by changing settings in the `browse.ini` file and the `[AddRelatedObject]` section. For newly created related objects, the same `browse.ini` file offers various settings (both default and per class). You can specify where the main node for such objects can be placed upon creation.

An interesting alternative is to use the `ezobjectrelation` and `ezobjectrelationlist` datatypes. The `ezobjectrelationlist` is a very powerful alternative to generic object relations as it allows categorizing related objects (create more than one `ezobjectrelationlist` attribute) and limiting the possible classes or related objects. You can even nest object relation lists by relating classes that in turn contain other object relation lists. The initial placement can again be configured in the `content.ini` file.

Workflows

A workflow can perform certain actions in the background (without user interaction) or require a user to perform certain tasks like approving an object before it can be published.

The workflow system of eZ publish provides the basics for implementing business procedures in electronic workflows. In the normal distribution of eZ publish, you have a few basic building blocks called **workflow event types**, but the workflow system can be extended with your own event types through PHP programming.

Triggers

Triggers launch the actual workflow chain. They are available for pre- and post-publishing workflows, as well as some shop functions. For each trigger, you can specify a workflow to execute. The list of triggers can be seen from the Triggers link of Set up, which becomes visible by expanding the Advanced section:

The screenshot shows the eZ publish administration interface. At the top, there's a header with the eZ publish logo, a search bar, and a 'Folder' dropdown. Below the header is a navigation bar with tabs: Content, Media, Shop, Users, Set up, and Personal. The 'Set up' tab is active, and the 'Advanced' section is expanded, showing a list of links: URL translator, URL management, RSS, Cache, Search stats, System information, and a dropdown for 'Advanced' which includes Classes, Templates, Sections, PDF export, and RAD.

The main content area displays the 'Trigger / List' section, titled 'Trigger list'. It contains a table with the following columns: Module name, Function name, Connect type, and Workflow. The table lists several triggers, all of which currently have 'No workflow' selected in the Workflow column.

Module name	Function name	Connect type	Workflow
content	publish	before	No workflow
content	publish	after	No workflow
shop	confirmorder	before	No workflow
shop	confirmorder	after	No workflow
shop	checkout	before	No workflow
shop	checkout	after	No workflow

At the bottom of the table, there is a 'Store' button.

You may think that you can have only *one* workflow per trigger. This is more or less true, but workflows can be parallel. In particular, the **multiplexer event type** is actually a hub for launching specific workflows depending on the object class and section (another use for sections!) Furthermore, you can specify an exclude-list of a group of users that cannot activate a specific workflow. For example, "approvers" who do not need to approve their own created objects need not activate approval workflows.

Workflow Events

A workflow consists of one or more workflow events. Currently, eZ publish comes with four predefined event types:

- Event/Multiplexer
- Event/Approve
- Event/Wait until date
- Event/Simple Shipping

The multiplexer event is a very versatile tool that allows multiple workflows to run for a given trigger. It provides a filtering on sections, classes, and users who are not the subject of a workflow, and then allows another defined workflow to run.

The approve event can filter on sections and define the users who can approve and the groups of users that do not need approval.

The wait-until-date event filters on classes, but relies on the presence of a date attribute of the class of the objects for which it enforces delayed publishing. You can specify more than one class and the date attribute on which publishing should occur. Normally, the publishing date of an object is set when a user issues a "send for publishing" action. In the wait-until-date event, you can override this with the actual publishing date defined by the class-specific date attribute.

The simple shipping event can be used in e-commerce or shop applications to add a shipping cost to an order. The values are not set in the workflow definition in the admin interface, but are obtained from appropriate values defined in the `workflow.ini` configuration file.

The following screenshots show the creation of a simple approval workflow for articles (a default class installed with the base distribution). First, a workflow is created consisting of a single approval event where the administrator user has to approve publishing (this is valid only if called from the pre-publish trigger). To do so, click on the Workflows menu entry under the Advanced section of the Set up tab. This will bring you to workflow groups. In a certain workflow group, select the New button to create your new workflow.

The screenshot shows the 'Workflow / Edit' page for a workflow named 'A simple approve workflow'. The interface includes a sidebar with navigation links like 'URL translator', 'URL management', 'RSS', 'Cache', 'Search stats', 'System information', and 'Advanced'. The main content area shows the workflow's name, its modification date, and a list of groups. Below the groups, there are sections for 'Events' and 'Users without approval'. The 'Events' section has a table with columns 'Pos', 'Description', and 'Type'. The 'Users without approval' section has a list of users and their counts.

Workflow / Edit

Editing workflow - A simple approve workflow

Workflow stored

Modified by Administrator User on 22/01/2004 12:22 am

Name
A simple approve workflow

Groups

eZ publish book ☐ **Remove**

eZ publish book

Events

Pos	Description	Type
1(1)	approve	Event/Approve

Editor
Anonymous User (10)
Administrator User (14)
test test (40)

Sections
Any
Standard section (1)
Users (2)
Media (3)

Users without approval
Any
Users (4)
Guest accounts (11)
Administrator users (12)
Editors (13)

Next, a multiplexer workflow is created. This will be attached to the pre-publish trigger.

The screenshot shows the 'Workflow / Edit' page for a workflow named 'Pre-publish approval chain'. The interface is similar to the previous one, but with different content. The 'Events' section has a table with columns 'Pos', 'Description', and 'Type'. The 'Classes to run workflow' section has a list of classes and their counts. The 'Users without workflow IDs' section has a list of users and their counts. The 'Workflow to run' section has a dropdown menu.

Workflow / Edit

Editing workflow - Pre-publish approval chain

Workflow stored

Modified by Administrator User on 22/01/2004 12:27 am

Name
Pre-publish approval chain

Groups

eZ publish book ☐ **Remove**

eZ publish book

Events

Pos	Description	Type
1(1)	filter on articles	Event/Multiplexer

Sections
Any
1-Standard section
2-Users
3-Media

Classes to run workflow
Any
1-Folder
2-Article
3-User group
4-User

Users without workflow IDs
4-Users
11-Guest accounts
12-Administrator users
13-Editors
42-Anonymous Users

Workflow to run
2-A simple approve workflow

An example of a delayed publishing workflow event is shown next. This should be attached to a new multiplexer event in the base workflow just created.

The screenshot shows the 'Advanced' configuration page for a workflow event in eZ publish. The left sidebar contains a navigation menu with options like 'Classes', 'Templates', 'Sections', 'PDF export', 'RAD', 'Extension setup', 'Workflows', 'Triggers', 'Translations', 'Packages', and 'Notification'. The main content area is titled 'Delayed publishing' and includes sections for 'Groups' and 'Events'.

Groups: A dropdown menu shows 'eZ publish book' with a 'Remove' button next to it. Below this is an 'Add group' button.

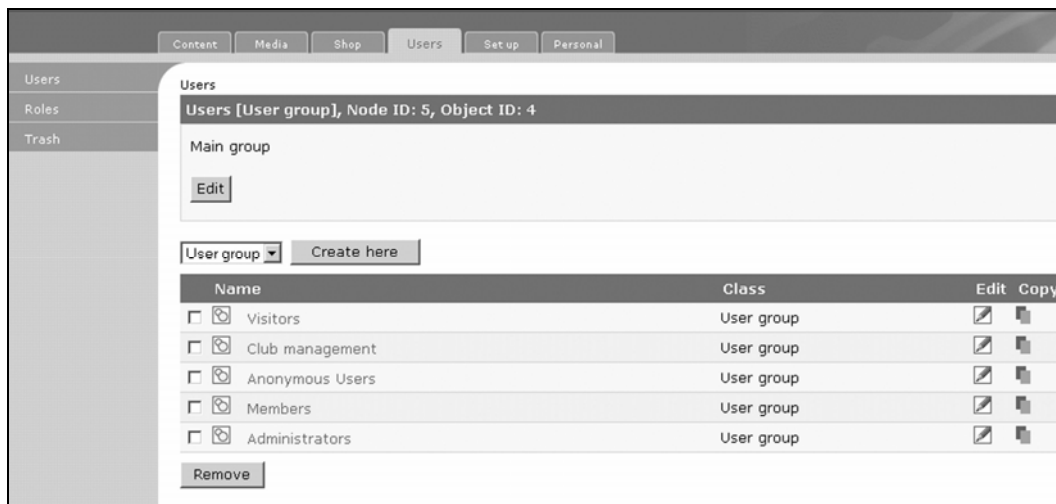
Events: A table lists events with columns 'Pos', 'Description', and 'Type'. The first event is at position 1(1) with the description 'Wait until' and type 'Event/Wait until date'. Below the table, there is a checkbox for 'Modify publish date' which is checked. Underneath, there are fields for 'Class name' and 'Class attribute name', and buttons for 'New entry', 'Remove selected', and 'Load attributes'. A list of classes is shown, including '11-Link', '12-File', '13-Comment', '15-Test class', and '16-Article with scheduled publish'. To the right, a list of class attributes is shown, including '156-Intro', '157-Body', '158-Thumbnail', '159-Enable comments', and '160-Publish on'. At the bottom, there are buttons for 'New', 'Event/Multiplexer', 'Remove', 'Store', and 'Discard'.

Permissions/Roles

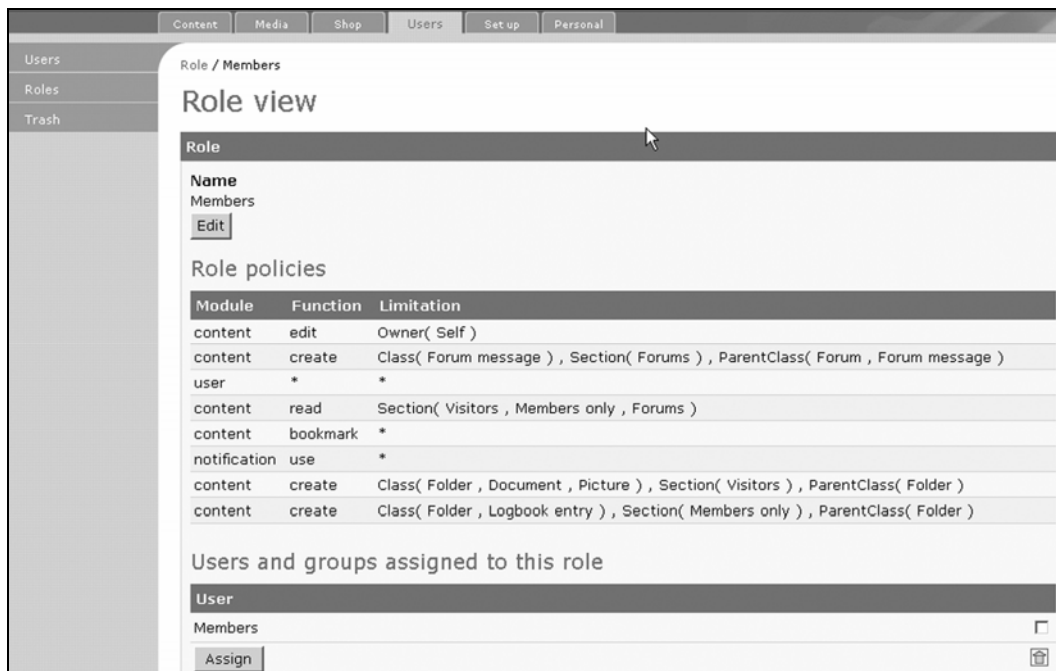
The permissions in eZ publish are based on roles. On one hand, this involves the definition of roles, and on the other hand, the users and user groups to whom the workflow is assigned.

Unlike traditional user-group systems, nesting of user groups is not supported in versions of eZ publish before 3.4. This is in spite of the fact that you can create hierarchies in the Users section of eZ publish. You can do this for other purposes (such as displaying an organization chart, for example), but for the role system, you needed to specify "flat" user groups—groups with user objects and without child user groups. Inheritance of user groups is supported in eZ publish version 3.4 onwards.

Users, user groups, and roles are created in the Users section of the admin interface:



Roles can be created in the Roles submenu, where you can define the policies associated with a role and the assignment to users and user groups. For example, here we see the creation of a typical role for users with limited editing capabilities.



These users will be able to:

- Log in
- Create (and edit) articles beneath folders
- Remove their own articles

Additionally, these power users are allowed to create bookmarks.

There needs to be an edit policy along with the create policy, because eZ publish generates a draft version upon content creation that is then edited.

Templates

The Template submenu in the Set up section can be used to edit and create override templates. By default, the templates available in the standard design directory are used.

However, for each template defined there (or in the directory tree below `design/<your design>/templates`), you can create templates that will override the default template for a given class, section, or other qualifier.

Some of the common qualifiers are accessible from the Templates submenu, but the complete power is only released by editing the `override.ini.append` file in `settings/siteaccess/<your siteaccess>`.

An advantage of template editing through the admin interface is that any cached version gets deleted after editing. This is not always true when using a standalone editor action within the file system when you edit the files directly instead of through the admin interface.

Upon first entry into the template edit system, you are presented with a list of common templates. This list is generic in the sense that it contains a predefined set of templates including hard-coded templates in the setup module of the kernel and the templates found in the `design/<your design>/templates` directory:

Template list

Most common templates

Template	Design Resource
/shop/basket.tpl	design/standard/templates
/node/view/execute_pdf.tpl	design/standard/templates
/node/view/full.tpl	design/standard/templates
/node/view/line.tpl	design/standard/templates
/node/view/pdf.tpl	design/standard/templates
/node/view/plain.tpl	design/standard/templates
/node/view/search.tpl	design/standard/templates
/node/view/sitemap.tpl	design/standard/templates
/node/view/text.tpl	design/standard/templates
/node/view/text_linked.tpl	design/standard/templates
/content/view/embed.tpl	design/standard/templates
/content/advancedsearch.tpl	design/standard/templates
/content/search.tpl	design/standard/templates
/fullscreen_pagelayout.tpl	design/standard/templates
/loginpagelayout.tpl	design/admin/templates
/pagelayout.tpl	design/admin/templates
/popup_pagelayout.tpl	design/standard/templates
/print_pagelayout.tpl	design/standard/templates

To actually work on templates, you will first need to select a type of template to work on. For example, select `/node/view/full.tpl`, and then select the siteaccess for which you want to edit or add override templates:

Template view /node/view/full.tpl

Default template resource **design/standard/templates**

plain_admin [Set]

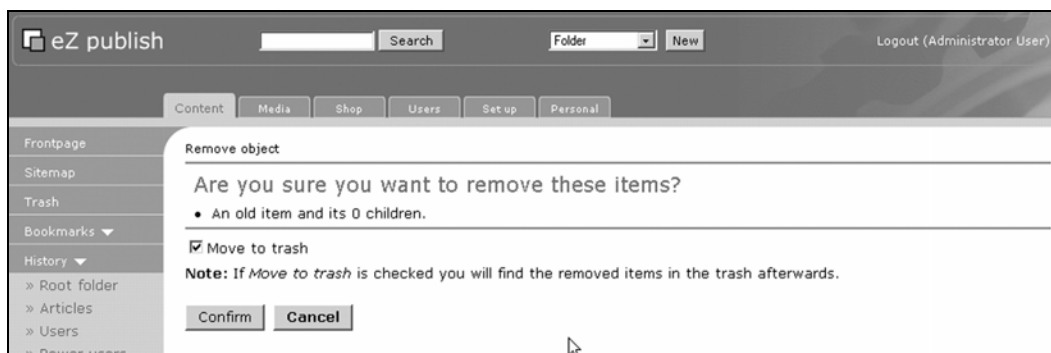
Override	File	Match conditions	Priority	Edit	Remove
media_folder	design/admin/override/templates/media_folder.tpl	class : 1 navigation_part_identifier : ezmedianavigationpart	1		
folder	design/admin/override/templates/folder.tpl	class : 1	2		
article	design/admin/override/templates/article.tpl	class : 2	3		
user_group	design/admin/override/templates/user_group.tpl	class : 3	4		
user	design/admin/override/templates/user.tpl	class : 4	5		
forum	design/admin/override/templates/forum.tpl	class : 6	6		
forum_message	design/admin/override/templates/forum_message.tpl	class : 7	7		
product	design/admin/override/templates/product.tpl	class : 8	8		
product_review	design/admin/override/templates/product_review.tpl	class : 9	9		
info_page	design/admin/override/templates/info_page.tpl	class : 10	10		
file	design/admin/override/templates/file.tpl	class : 12	11		
comment	design/admin/override/templates/comment.tpl	class : 13	12		

More Administrator Functions

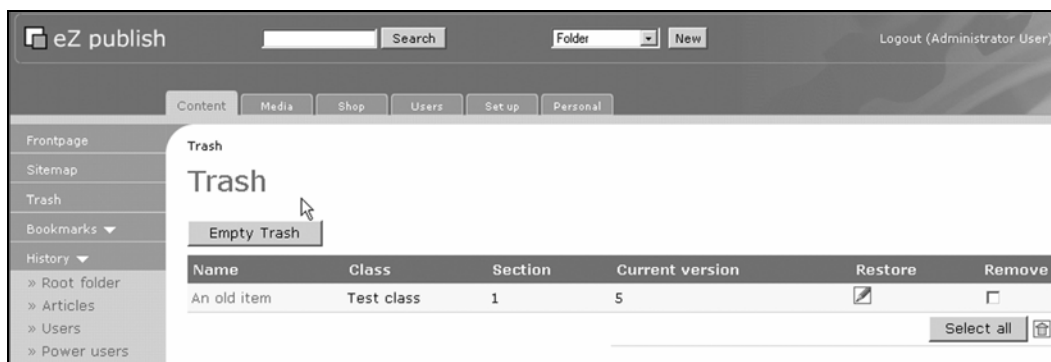
In this section, we will look at some of the common functions found in the administration area of eZ publish.

Removing and Restoring Objects

You can remove objects in the admin interface from the Content section. Since version 3.3, you have the option to put objects into a trashcan so they can be restored later or to remove an object permanently. Here we see the result of a remove action:



And here is the trashcan view:



From the trashcan you can restore objects into their original location by default.

URL Translation

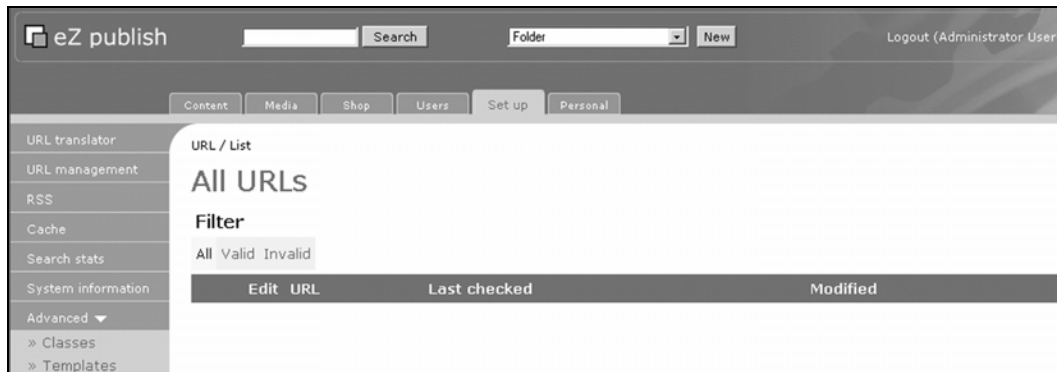
eZ publish provides a mechanism for URL translation. This can be handy for the construction of easy-to-remember URLs like `http://mysite.com/about`, which in

reality points to an article deep inside the site node tree. You can also use this to translate entire hierarchies into a new location or handle old URLs that are no longer valid.

When moving objects around in your eZ publish site, you strictly do not need to retranslate locations; eZ publish will redirect to the new location automatically.

URL Management

Every URL in your eZ publish site, entered either from the `ezLink` attribute or as an external link inside an `ezxmltext` attribute, is stored separately in the database. When properly set up, the scheduled cron jobs will check all these URLs for validity. This is one of the unique features in eZ publish and the admin interface provides access to this functionality as it gives a list of valid and invalid URLs used in your site:



For each URL, you can also see the objects that make use of the URL. This can help you decide to edit the objects or simply replace the invalid URLs with new valid values.

If your server is behind a firewall that blocks outgoing HTTP requests, you need to modify the cron job for URL management to make use of a proxy or contact your network administrator to allow outgoing HTTP traffic.

RSS Export and Import

eZ publish adheres to major standards and RSS is one of them. You can do both RSS exports and imports from a variety of sources. The current implementation supports RSS import of versions 0.9, 1.0, and 2.0. RSS exports can be configured for RSS version 1.0 or 2.0. For exports, you can define the mapping of the children of a particular node (one level of depth and limited to five items). The following screenshots demonstrate RSS import and export respectively.

eZ publish Search Folder New Logout (Administrator User)

Content Media Site Users Set up Request

URL translator
URL management
RSS
Cache
Search stats
System information
Advanced ▾
 > Classes
 > Templates
 > Sections
 > PDF export
 > RAD
 > Extension setup
 > Workflows
 > Triggers
 > Translations
 > Packages
 > Notification

Really Simple Syndication

RSS Import

Title:

URL:

Destination path:

Imported objects owner:
Administrator User

Class:

Title:

URL:

Description:

Active:
☒ Active

Content Media Site Users Set up Request

URL translator
URL management
RSS
Cache
Search stats
System information
Advanced ▾
 > Classes
 > Templates
 > Sections
 > PDF export
 > RAD
 > Extension setup
 > Workflows
 > Triggers
 > Translations
 > Packages
 > Notification

Really Simple Syndication

RSS Export

Title:

Description:

Site URL:

Image:

RSS version:

Active:
☒ Active

Access URL:

Note. Each source only fetch 5 objects from 1 level below.

Source 1

Source path:

Class:

Title:

Description:

For both import and export, you need to map the RSS fields to the eZ publish class and attributes. For RSS import, the default link class is a good candidate. Imported RSS items are stored in a node tree location that can be specified in the RSS configuration admin interface.

Cache Administration

eZ publish is a powerful application, but is also resource intensive. To minimize the use of resources (CPU and memory), cache mechanisms are implemented for the major parts of the underlying kernel functions and libraries. Roughly, caching is active in for the following operations:

- Content views
- Template compiling
- INI file caches

You can use the cache administration screen to clear any of the sections of the cache, or even make a fine-grained selection to clear, such as the template block cache:

The screenshot shows the 'Cache admin' interface. On the left is a sidebar with navigation links: URL translator, URL management, RSS, Cache (selected), Search stats, System information, and an 'Advanced' dropdown menu containing Classes, Templates, Sections, PDF export, RAD, Extension setup, Workflows, Triggers, Translations, Packages, and Notification.

The main area is titled 'Cache admin' and contains a section 'Cache collections' with the instruction 'Click a button to clear a collection of caches.' Below this are four buttons: 'All caches', 'Content caches', 'Template caches', and 'INI caches', each preceded by a label: 'All caches.', 'Content views and template blocks.', 'Template overrides and template compiling.', and 'INI caches.' respectively.

Below the buttons is a table with the following data:

Name	Path	Selection
Content view cache	content	☐
INI cache	ini	☐
Codepage cache	codepages	☐
Expiry cache	expiry.php	☐
Class identifier cache		☐
Sort key cache		☐
URL alias cache	wildcard	☐
Image alias		☐
Template cache	template	☐
Template block cache	template-block	☐
Template override cache	override	☐

At the bottom right of the table area is a button labeled 'Clear selected'.

Normally, cache clearing is necessary during development or after upgrades that you should perform only in a development environment instance of your site. However, small changes on a live (production) site gain from the cache interface. The fine-grained control is especially useful for live sites.

Caches should be cleared after major updates in design (templates) and system upgrades (eZ publish upgrades). Make sure that you also clear the caches from PHP accelerators when executing system upgrades, as these can become corrupt due to changes in the PHP classes used.

Search Stats

While eZ publish by default does not keep track of the objects visited, it does keep statistics on search phrases and words. This helps you determine if the taxonomy of your site matches the average expectation of your target audience. The Search stats page shows a view of these statistics and also allows the clearing of statistics gathered this far:

Search stats

Search statistics

Most frequent search phrases

Phrase	Number of phrases	Average result returned
test	1	1.00
new	1	0.00
news	1	1.00
camera	1	1.00

Reset statistics

System Information

The system information page shows details of the current setup, including PHP version, PHP extensions, and other important settings. This is useful in determining if the PHP configuration, HTTP server (mostly Apache), and other items meet the requirements of eZ publish.

Cache

- Search stats
- System information
- Advanced ▼
 - Classes
 - Templates
 - Sections
 - PDF export
 - RAD
 - Extension setup
 - Workflows
 - Triggers
 - Translations
 - Packages
 - Notification

Site: 127.0.0.1/index.php

Version: 3.3-2 (3.3)

SVN revision: 4764

Extensions:

PHP

Version: 4.3.4

Extensions: standard, bcmath, calendar, ctype, com, ftp, mysql, odb, overload, pcre, session, tokenizer, xml, wddx, zlib, apache, ldap, mbstring, Turck MMCache

Safe mode is off.
 Basedir restriction is off.
 Global variable registration is off.
 File uploading is enabled.
 Maximum size of post data (text and files) is 8M.
 Script memory limit is .
 Maximum execution time is 60 seconds.

PHP Accelerator

Name: Turck MMCache

Version: Could not detect version

The PHP Accelerator is enabled.

Database

Type: mysql

Server: localhost

Database: plain

Connection:

retry: 0

Section Setup

In the Sections page, you can define or remove section definitions. Here we see the basic (minimum) sections as defined for the plain demo package:

eZ publish Search Folder New Logout (Administrator User)

Content Media Site Users Set up Personal

Sections

Section list

ID	Name	Edit	Assign	Remove
3	Media			
1	Standard section			
2	Users			

New

When adding sections, you can specify a navigation part that can be used in templates. For example, the admin interface itself makes use of this to map certain parts of the content to other navigation tabs. Normally, you do not want to do this except for assigning a gallery of images and multimedia parts of your node tree to the Media tab.

A common error in eZ publish administration is the deletion of the Users section by accident. This leads to the situation that the Users tab no longer displays the left-hand menu for users and roles. To restore this functionality, you need to create a new section that is assigned to the users root node and has Users as the navigation part. The integer value known as the section ID is not important here.

PDF Export

eZ publish not only provides web content through normal web browsing or RSS feeds, but also delivers content as an Acrobat PDF file. This is done in the PDF export submenu where you can define several PDF export configurations for parts of your site. Due to the heavy processing involved, you should limit the generation of PDF files of your content to relatively few objects (at least in version 3.3 of eZ publish).

The following screenshot shows an example entry for PDF export where you can define the node or node tree, classes, introductory page elements, and the way of publishing (either direct download or a cached PDF file).

The screenshot displays the 'PDF Export' configuration page in the eZ publish administration interface. The left sidebar shows a navigation menu with the following items: URL translator, URL management, RSS, Cache, Search stats, System information, and an expanded 'Advanced' section containing Classes, Templates, Sections, PDF export (the current selection), RAD, Extension setup, Workflows, Triggers, Translations, Packages, and Notification. The main content area is titled 'PDF Export' and contains the following fields and options:

- Title:** A text input field containing 'Book of my site'.
- Display frontpage:** A checked checkbox.
- Intro text:** A text input field containing 'This is an off-line version of the site including all the articles'.
- Sub text:** A text input field.
- Source node:** A text input field containing 'articles' and a 'Browse' button.
- Export structure:** Radio buttons for 'Tree' (selected) and 'Node'.
- Export classes:** A list box containing 'Folder', 'Article' (selected), 'User group', 'User', 'Image', 'Forum', 'Forum message', and 'Product'.
- Export destination:** Radio buttons for 'Export to URL' (selected) and 'Export for direct download'. The 'Export to URL' option has a text input field containing 'mysite.pdf'.
- Export:** A button at the bottom of the form.

Rapid Application Development (RAD)

The RAD menu currently provides tools for generating skeleton PHP files for the creation of new template operators or datatypes. It is expected that later releases will include more options.

Extension Setup

You can activate third-party extensions such as the Online Editor in the extension setup menu. These need to be enabled in the appropriate INI files. Most extensions come with an explanation of how to do this, and any requirements specific to enabling the extension.

Packages

With the Packages menu, you can install or remove packages in the eZ publish native format. Currently supported types are class packages and design packages (CSS files). This is useful when you create several classes that you may need on other installations. Third-party submissions from the eZ publish community may come in this format for convenient installation.

When importing a package with new classes, make sure you have all the included datatypes installed. If not, the package import will fail, leaving the `contentclass` and `contentclassattribute` tables in a **dirty** state.

Notification

The notification system can be run *ad hoc* from the notification section. The functions provided here are used mainly during development to test the proper functioning as they should normally be executed from a scheduled cron job. It can be used as a workaround for installations that do not allow access to a cron system or scheduler during development.

Personal

The Personal tab provides some useful functions for individual users (provided that you have created and assigned the necessary policies). It provides:

- Drafts for listing objects in draft mode by the users currently logged in.
- A pending list that details objects that have a pending workflow event, such as approval or delayed publishing.
- Notification settings for sending e-mails about new or updated objects (somewhat limited in version 3.3, but this will improve with later releases).

- Bookmarks for storing nodes of interest at the server level (independent of the browser or computer that the current user is working from).
- Collaboration—currently this holds the items for which an action is required by the currently logged-in user, such as approvals.
- Password management for changing the password of the current user. The administrator can also do this in the Users section for any user.

Shop

The Shop tab provides some functions for e-commerce (shopping) like different types of taxes or discounts to apply for certain objects and user classes. When users add items to their baskets, the admin interface provides simple handling of shopped items.

Creating an Example Site

This section is all about the first few steps that you need to take to build your own sites with eZ publish. To jumpstart, you should install the plain package that comes with the default distribution of eZ publish.

We begin with an overview of typical requirements of our project: setting up a portal for a nature photography and astronomy club website.

The portal for the nature photography and astronomy club should provide the following feature set:

- A public part with documents, pictures from members organized in galleries, discussion forums, calendar of activities, logbooks, news (with subscription service), and a few personalization features (for registered users)
- A private part for the club management for statutory documents, meeting reports, actions, and private discussions

Creation of Basic Classes

Now that you have learned about the powerful features of eZ publish to accommodate whatever high-level data model and structure you want for your website, a little analysis of the requirements soon brings forward a number of solutions. Indeed, with eZ publish you often have more than one way of accomplishing your goals.

Using the example website, the requirements can be synthesized as follows.

Documents

The term document means different things, yet we want to limit the number of document types for our website. Roughly speaking, we want both structured documents, where the different parts such as title, introduction, body, keywords are separated, and legacy documents in the form of PDF files or typical word-processing documents. This is possible if we use a super-document class consisting of the following elements:

- Title (text line)
- Intro (XML text field)
- Body or summary (XML text field)
- A file as attachment (binary file)
- A keyword list (keyword datatype, as an automated form of related documents)
- Miscellaneous related items (object relation list)
- A flag for allowing/disallowing comments (as with the default provided article class)
- A flag to signal that a document should appear as a news item on the home page

Another type of document is a logbook—a kind of dedicated weblog. This is in fact a collection of logbook entries linked to each user registered with our site. The logbook itself will be a simple folder object. A logbook entry will consist of a simple list of elements:

- A subject (text line)
- A body (XML text field)
- A list of related items published elsewhere on the site (object relations list)

Images

Images won't be just an image with a caption. Our nature photography and astronomy friends are eager to provide information on the equipment they used, location, date and time, and perhaps a few other details. Since this is not a book on astronomy, we'll nevertheless limit the number of attributes.

After some deliberation with the website project team (the webmaster and his daughter in this case), the image class will consist of:

- A title (text line)
- A description (XML text field)
- Image file

- Caption to display with each picture (text field)
- Equipment used (XML text field)
- Date and time (datetime field)
- Location (XML text field)
- Keywords

Discussion Forums

For the forums, we basically need a list of topics, sticky forum messages (the ones that are displayed at the top of each forum containing important information), and a forum folder. The folder will be just the standard folder object as delivered with a plain eZ publish distribution, consisting of a title (text line) and a description (XML text field). The forums are also folder-like objects, but nevertheless we'll create a new class for them adding an icon as a new attribute. Messages or topics (a top-level message) will consist of the following attributes:

- A subject (text line)
- A message body (plain text field)
- The type of message (selection datatype) to distinguish between a question, a remark, important news, and so on

Calendar of Activities

Activities can be of different types, but nevertheless we want them to be implemented with only one class. A calendar item therefore consists of the following attributes:

- Subject (text line)
- Description (XML text field)
- Location (XML text field)
- Start date and time (datetime)
- End date and time (datetime)
- Keywords (keyword)
- Category (selection)
- Related items (object relation list)

The navigation will also need a calendar; the kind of component you may be familiar with from groupware applications.

Personalization

With respect to personalization, we want each registered user (member if you wish), to be able to have his or her own mini-site with details on their level of skills, equipment, and other information. In addition, we want to offer some preferences and access to notifications, a draft space, and user-interface preferences. We can implement this under an umbrella (a folder) object with subfolders containing user-specific details.

Miscellaneous

We will need one more class: a generic object with only a name. We will use objects created for hosting special functionality, such as a home page, in the templates later on. Our dummy class will be:

- Name (text line).
- For each of the nodes associated with this class, we will make a dedicated template overriding the default. It will hold things like the entry point of My site with preferences, statistics, and more.
- Furthermore, we also want a links or bookmarks section of general interest (not to be confused with the bookmarks functionality in eZ publish). For this, the standard link class will be used, which consists of a title, a description, and the link itself with a URL and the display name (the thing you generally put between `<a . . . >` and `</a>`).
- Finally, we need comments that will be added by members. This will be done in the templates, which we look at in the next chapter.

Creating Classes

Creating a class is left as an exercise for you by using the default admin interface delivered with eZ publish. The mechanics of this has been covered in some detail in the *Creating Content Classes* section.

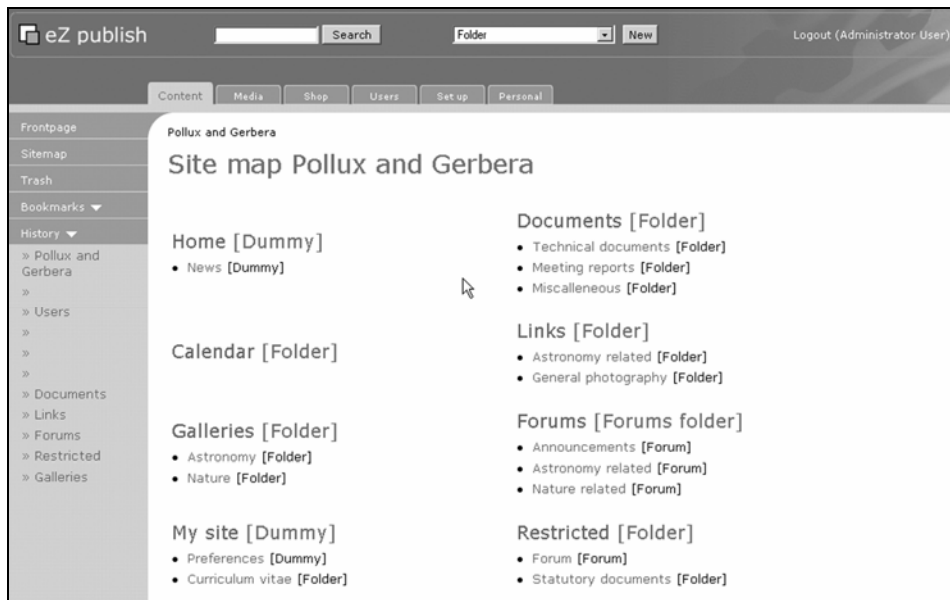
Taxonomy or Structure

The site structure will be straightforward, and the visual experience will be made mainly through the use of powerful template functions provided by eZ publish. When viewed as a tree-like structure, the taxonomy will consist of:

- Home (a dummy object) aggregating news and other items
- News (show all the news items with a navigator for archived news)
- Documents
- Technical documents

- Meeting reports
- Miscellaneous
- Calendar of events
- Links
- Astronomy related
- General photography
- Galleries
- Astronomy
- Nature
- Forums
- Announcements (by the club management)
- Astronomy related
- Nature related
- My site (personal stuff)
- Restricted part for the club management
- Forum
- Statutory documents

Here we see the sitemap view in the admin interface (the sitemap template has been modified to show the class name of the objects).



Users and Roles

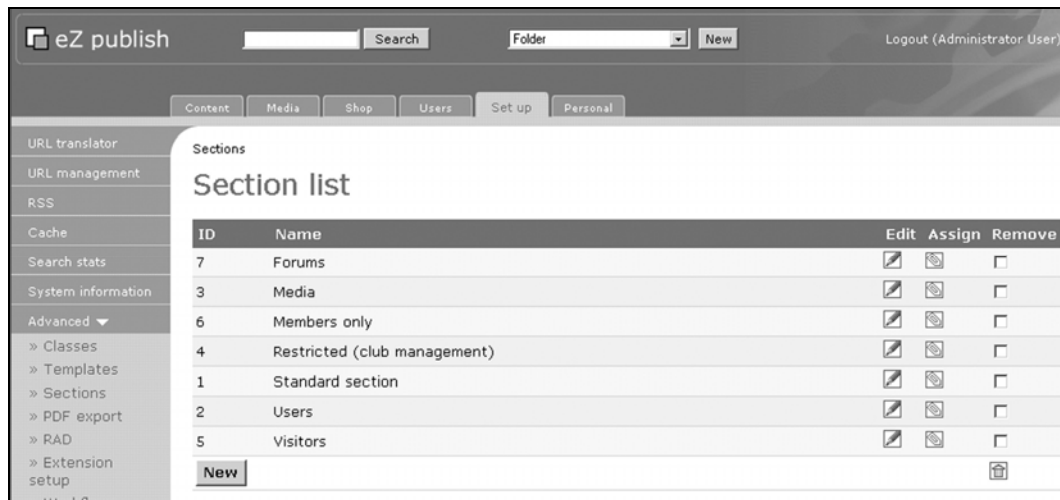
For the roles settings, we want users (members) and the general public to be able to read everything except for information in the restricted parts. Regular members should also be able to create new forum messages and topics, and submit documents and images (which would be reviewed by the club management). Self-registration will be disabled—every member will be created by the club management. For example, they could grant membership on the condition that applicants have paid their annual membership fee.

Sections Setup

To make it easier to implement permissions and roles, we will assign sections to the various parts of the site. By default when new folders are created, they inherit the section from the parent (or top-level) element, usually the Standard section. For our purposes, we can make the following sections and assign them to subtrees in the taxonomy described earlier:

- A restricted section for club management
- A section for forums where only members can post
- A section where each member can view the details of other members, but which is not accessible to the broad public (anonymous users in eZ publish)
- A visitors section for all users, including anonymous users, which is assigned to the top-level elements (Home, Documents, Calendar, Forum)

Here we see the result of creating these sections:

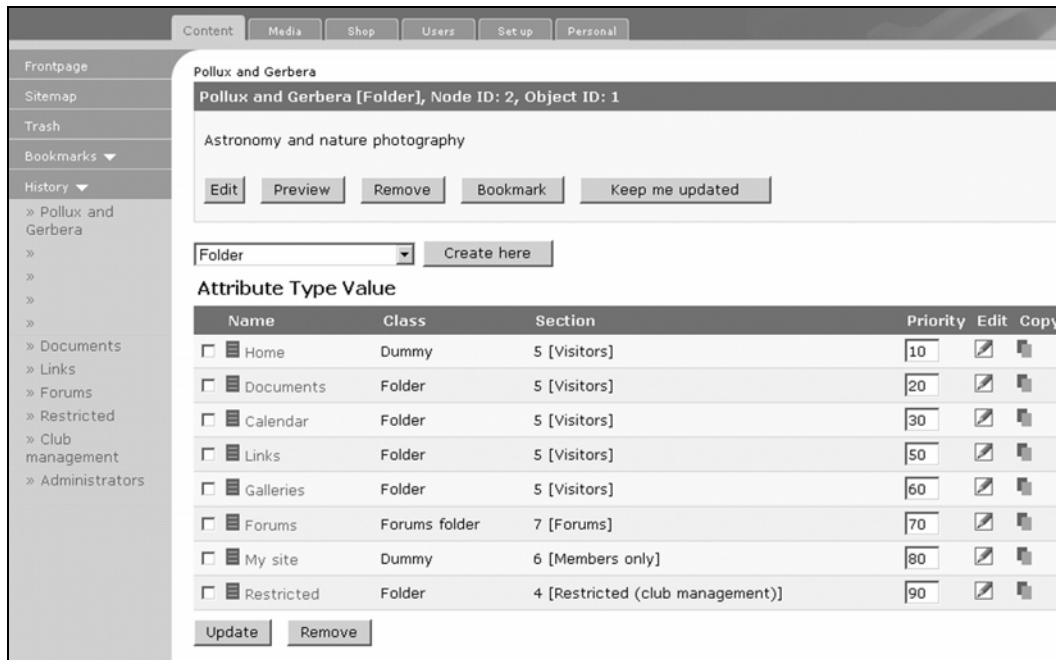


The screenshot shows the eZ publish administration interface. The top navigation bar includes the eZ publish logo, a search bar, a folder dropdown, and a 'New' button. The user is logged out as 'Administrator User'. The main content area is titled 'Sections' and 'Section list'. It contains a table with the following data:

ID	Name	Edit	Assign	Remove
7	Forums			
3	Media			
6	Members only			
4	Restricted (club management)			
1	Standard section			
2	Users			
5	Visitors			

Below the table is a 'New' button and a small icon for creating a new section.

The following screenshot is the result of their assignments:



In this screenshot, the template from the admin interface was slightly modified to show the section-access details in square brackets alongside the section ID.

User Groups

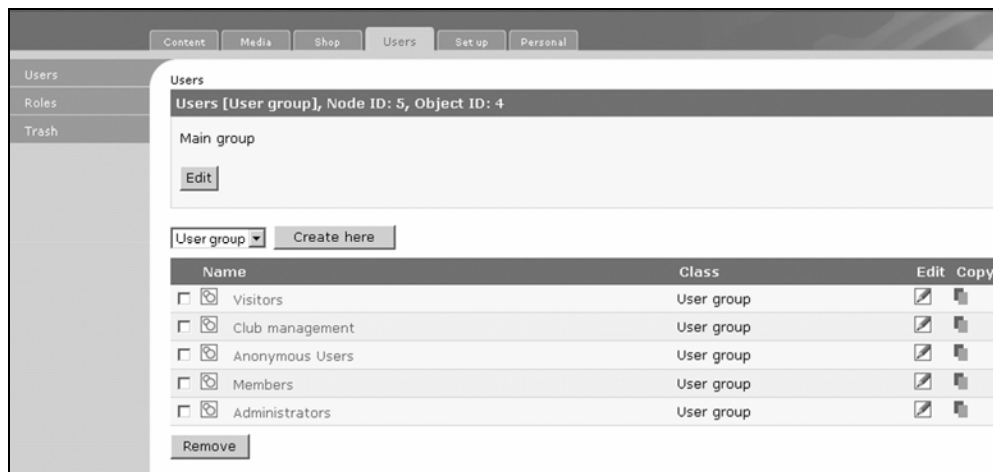
Basically, there are four categories of users:

- Anonymous users
- Visitors (non-members registered with the site)
- Members (assigned by the club management)
- Club management

For each of these categories, you need to create user groups. The default group for non members will be the visitors group. This has to be set in the settings/override/site.ini.append.php file in the [UserSettings] section with the DefaultUserPlacement parameter. For example, in the tutorial site, the node ID of the Visitors group is 12, so the configuration line should look like:

```
File: settings/override/site.ini.append.php
[UserSettings]
...
DefaultUserPlacement=12
...
```

Here are the final user groups:



Roles and Role Assignments

For clarity, each type of user (defined in their separate user groups) is assigned a role definition.

Anonymous

An anonymous user gets a content/read policy for access to the visitor section and has the permission to log in. You can also disable policy checking in the [RoleSettings] section in settings/override/site.ini.append.php by specifying PolicyOmitList[]=user/login (default setting at the time of writing).

The role view of the anonymous role is shown in the following screenshot:



Visitors (Registered Non-Members)

Registered users have almost the same permissions as the anonymous user, except that they have two additional permissions—creating bookmarks and notifications.



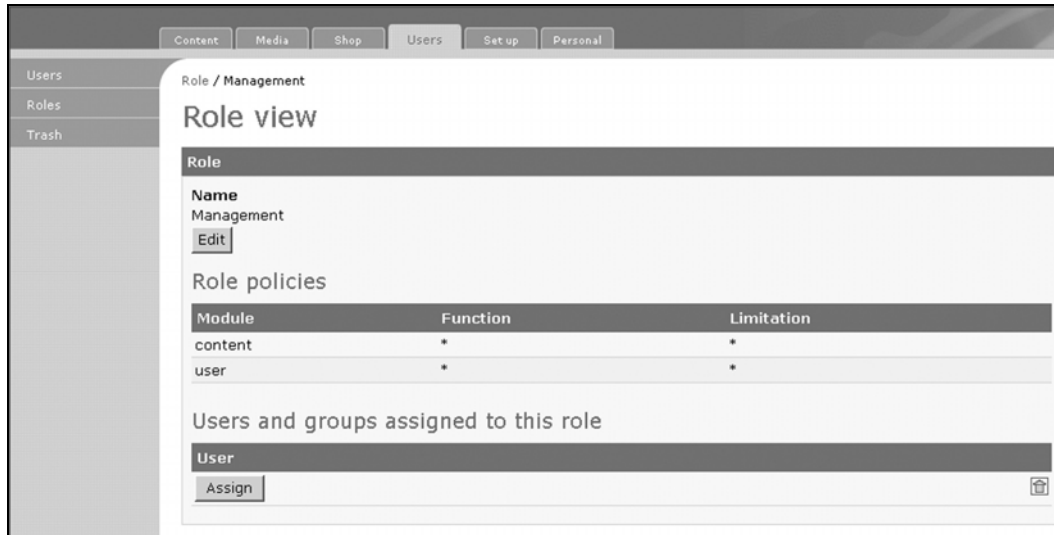
Members

Members are granted the ability to create documents, upload pictures, and keep their logbooks in addition to the policies defined for visitors.



Club Management

The club management can do almost anything with the content and users, so the policies are rather simple:



Summary

In this chapter, we have seen how to use eZ publish for content management. We looked at the fundamentals of an eZ publish site and content handling within it, including content classes and objects, versioning, and how eZ publish displays your content.

We moved on to the administrative interface of eZ publish, and how it is used to add content to your site. We looked at the default datatypes that are available with eZ publish from which can construct your content objects, as well as how the other main concepts of eZ publish are realized in the administrative area.

We concluded the chapter with a tutorial on the basics of creating a sample astronomy and nature photography site, and walked through the basic classes and roles required by the site.

In the next chapter, we look at the templating system of eZ publish, which is at the heart of content display in eZ publish.

3

Displaying Content with eZ publish Templates

eZ publish comes with a powerful template engine that is capable of presenting the content in various output formats, including XHTML and PDF. Template files can also be used to create functionality that is sometimes described as **portlets** (portal components) in other content management systems.

You already know how eZ publish can be used as a content management system from a pure content perspective. The next layer for gaining knowledge is the template system and how it can be used to extract and present content to the user. The caveat is that the syntax of the template system means another language to learn.

In eZ publish, every page is generated dynamically, based on templates that are processed to add the content. This is because the architecture of eZ publish separates design and presentation logic (including internationalization) from content and processing, which happens mainly in the kernel modules.

Templates are generally a mix of XHTML and the eZ publish template language, providing access to the data (in the database or file system) and functions of the underlying application and content management framework.

In the next section, the overall idea of the template system is explained. The subsequent sections go into the details and show you how to use the template language.

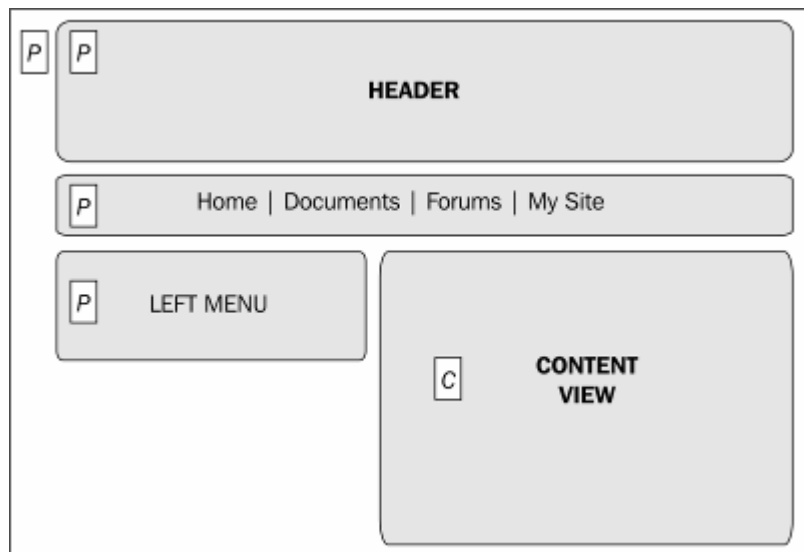
Principles

The eZ publish template system is designed in a hierarchical fashion, where some first-level templates (`pagelayout.tpl`) control the rendering of more specific blocks like the content views and navigation structures (menus). Content views are rendered through templates, which can be tied to specific classes and view modes. The datatypes that make up a class are also controlled by individual templates. Further down the template hierarchy, even individual elements are controlled by templates (for example, the

<header> tags in XML text fields). Almost all templates can be overridden by custom templates, depending on the section, class, depth in the node tree, and so on.

Page Layout and Content Views

When you visit the main eZ publish site or most of the sites created with it, you are presented with a page containing navigation elements (such as top-level menus), polls, and content, such as article listings, image galleries, and so on.



eZ publish distinguishes between page layout templates and content view templates. Page layout templates are used to present the user with a global page layout. This includes everything except the main content. The main content referred to here is what is actually requested by the URL, and displays along with the navigation elements to assemble the whole page.

However, because of the power of eZ publish, the distinction between page layout elements and content presented here can be blurred by your specific needs, imagination, and creativity.

The template system is component based, which means that a template generally calls other template files for common parts or when you need a particular functionality, or intrinsically by the hierarchical organization of templates from the top level (page layout) to individual objects, object attributes, and the XML text tags of XML text attributes.

Page Layout

The main template that is always called is the `pageLayout.tpl` template. This template will be used to display the overall page layout, `<head>` elements such as cascading stylesheets, the `<body>` section top-level menus, navigation parts, a search box, etc.

The page layout template can be overridden by specific instances. For example, the 'home' page layout may be different from the page layout used for presenting a catalog of articles or products.

You can also define additional page layouts for a printer-friendly version or for pop-up windows. These are called by prepending the normal URL part with `/set/layout/<your pageLayout.tpl override>/`.

Content Views

View templates are mostly associated with the content views. The content view is actually a presentation of the content or page requested by the user through the URL. For example, `http://www.example.com/content/view/full/2/` refers to viewing the content in full view mode of node 2.

When you activate 'nice URLs', the URL actually does not contain a view mode. The full view mode is implicitly tied to the nice version of the URL.

There are more standard view modes that can be called in the URL, such as `http://www.example.com/content/view/sitemap/2/`, where a sitemap View template is called to present the content of node 2.

Alternatively, you can create your own view modes and call them with a name you see fit (and which is not yet defined). The only thing you need to do is to create a file with your specific view name in the default directory for View templates:

```
design/mydesign/templates/node/view/
```

For example, if you want to have a view mode that displays more information than the full view template, you can create a template called `extended.tpl`, which could add, for instance, the audit trail (version history), the date when the content was last modified and by whom, related objects, and so on.

These custom views can further be subjected to the template override system, effectively adding another dimension to the possibilities of the eZ publish template system.

A second class of often-used view templates corresponds to view modes as used in lists (the line view) or when embedded within XML text attributes (the embedded view).

Like page layout templates, content view templates may be overridden for specific classes and also depending on which section is active or which node is being called.

Attribute Templates

The templates in eZ publish are further broken down in components for object attributes; these constitute the **attribute view** templates. They are normally used for abstraction in the display of normal view templates for nodes. For example, the default view template for the full view just loops over all the available attributes and calls a generic template function (`attribute_view_gui`), which takes care of the proper display, depending on the datatype. The attribute view templates are located in the `/design/standard/templates/content/datatype/view/` directory.

Attribute templates can also be overridden for specific classes, nodes, and sections.

Template Modularization

Template code can be reused by placing it in specific template files. These can be included in other template files by using the `include` template function. Along with this function, you can specify parameters to pass to the included template. In this way, templates can become the equivalent of small applets that take input parameters.

Some typical examples of these are:

- `<head>` elements for specifying style sheets
- A calendar widget
- Navigators
- Menu/navigation blocks

StyleSheets and Images

To adhere to the design standards of eZ publish, most of the style information should be put in CSS files, leaving only pure XHTML and template code in the template files. CSS files should be placed in `design/mydesign/stylesheets`.

If you need to use images that are not part of the content of your site (such as navigation icons or logos) in your template, you should place them in the `design/mydesign/images` directory.

Edit Templates

For a lot of websites, the edit functionality you offer to the end user may be quite limited. However, for intranet portals and other highly collaborative sites, you may not be content with the default edit functionality of the admin interface. To get around this, you can

create **edit templates** to allow specific users to modify part of the content and offer tailored edit interfaces that are integrated seamlessly within the same site design, coupled to the normal site access.

Templates and Caching

As the template mechanism is used for abstracting the data from presentation, the processing involved is considerable and the page-processing time is typically an order of magnitude larger than spaghetti-style PHP with HTML embedded or vice versa. Since version 3.1, the template engine also includes a template compiler that converts the eZ publish template code into native PHP code. This code is stored in the `var/cache` directory of your installation (the most daring among you are invited to look at the generated PHP code: it is impressive). You can enable or disable template compiling in your `site.ini.append.php` files.

Not only the template code, but also the content retrieved from the database is cached as PHP code. This is called view-mode caching and actually combines data (node structures converted to multidimensional arrays) with template output (normally XHTML).

Cascading and Overriding Templates

When you learned about configuring your site in Chapter 1, the structure of design directories was introduced. The template system partly depends on this structure to determine the template file to use. Along this directory-traversing algorithm, you can override the templates depending on specific conditions defined in the `override.ini.append.php` file.

In this way, you can specify a certain template to be used according to a specific node, the depth in the tree, the class of the object, or some other parameters (explained later in this chapter).

Working with eZ Publish Templates

After this general overview of the principles, the next step is to explore the various aspects of the template system in more detail.

Overview

The main prerequisite for the template system to work properly is that the configuration parameters should be correctly specified, according to the instructions in Chapter 2. Where appropriate, some details on configuration settings are treated with specific topics.

Where Does the Content (Data) Come From?

To manipulate the content for display in templates, it is important to know where the content (or part thereof) comes from. The following list should provide a decent idea:

- **Navigation elements, such as menus, or banners:** Navigation elements could be hard-coded in the page layout template, but they are, in general, tied to the site structure or taxonomy of your site. Therefore it is sensible to fetch the site structure dynamically and use the results to build menus and other navigation elements.
- **Applets where content is explicitly fetched like polls, news, or RSS feeds:** Applets are generally built using applet-specific templates and called with certain parameters, which are usually nodes somewhere in your site.
- **The `module_result.content` as the main content delivered from the requested module or function:** The main result of the module/function called is handled by specific content templates, but you can use a few elements from the result set for other purposes.
- **Additional content implicitly fetched by the main module/function result(s):** Finally, the main content, as it is being handled by the view templates, is a rich object structure that you can use where appropriate.

The following sections discuss the basic syntax of the eZ publish template system. As stated before, the template system employs an eZ publish-specific language which you can use in your own templates. We are now actually getting into the template files, so buckle up!

Comments

It is good practice is to add comments to your template files. Although comments generate a little overhead in parsing the template files, you should not be economic in their usage. Document the objectives and implementation for your custom templates in terms of features, methods, and expected outcome.

Comments are enclosed between `{*` and `*`. The content between these delimiters is never sent to the client. Actually, it is ignored even by the compiled template code.

```
{* this is a comment *}
{* This is a
multi line
comment with a portion of code commented out:
{let fruit=array(apple,orange,strawberry)}
{section loop=$fruit}
{$:item}
{/section}
{/let}
*}
```


If you really want or need comments sent to the client, you can use XHTML comments. However, be aware that these comments are also parsed by the template subsystem, so you need to make sure that there are no clashes with the template syntax.

Also, be careful with JavaScript in templates, especially the `{` and `}` characters, which are treated as 'block start' and 'block end' by the template parser. This means you cannot input them directly in the template code to get a `{` or `}` on the web page. Instead, you must use the `{ldelim}` and `{rdelim}` functions, as shown in the following example:

```
{* use of the left and right delimiters *}
Hi, the next piece of template code uses {ldelim} and {rdelim} tags
```

This produces the following:

```
Hi, the next piece of template code uses { and } tags
```

Note that the comment block does not appear in the output!

To ignore larger blocks of template code with potential conflicting delimiters, you can also use the `{literal}` construct. For example, to make sure JavaScript code blocks will work, surround them with this tag, as shown in the following example:

```
{literal}
<script language="JavaScript">
<!--
function MyJavaScriptFunction ( Param ) {
    Param=1
}
// -->
</script>
{/literal}
```

Variables

Variables are containers for various types of content, and are set either by operators and functions of a module, or explicitly inside templates.

The variables that you define inside a template file will have their scope for that particular template.

Setting and Modifying Variables

To set or modify variables, use the constructs `{let ...}`, `{set ...}`, or `{default ...}`.

`{let}` and `{default}` declare variables, and with `{set}`, you can modify them further. `{let}` and `{default}` require closing equivalents `{/let}` and `{/default}` respectively, which also un-set the variables they defined.

```
{let  a=3
      b='a string'
      c=false() }
    {$a} times {$b} is {$c}
    {set c=true()}
    {$c}
{/let}
```

This code will output the following:

3 times a string is 1

{default} is special, since it will assign a value to a variable unless it is already defined by a parameter passed to the template where {default} is used. This helps make templates more robust and modular.

To assign the result of complex template code to a variable or for larger text fragments, variables can also be set with {set-block}. The syntax is:

```
{set-block name=<name> scope=<scope> variable=<myvariable>}
{* things you want to assign to, including further eZ
                                     template expressions*}
{/set-block}
```

The name parameter is optional and introduces a new namespace (discussed further in the sections on variables and namespaces). scope can be global, root, or relative, and variable is the identifier to use. The scope parameter makes it possible for a template to return different text fragments, as opposed to a single output block. This is used, for example, in e-mail templates where the subject and body text are set separately.

With the global keyword for the scope parameter, you can even set variables that live in the global namespace, which is mostly outside of the current template. For example, you can define a variable called extra_javascript in the <head> section of your pagelayout.tpl file, in which you can set blocks of JavaScript from within a view template. This is handy for reusing JavaScript files or adding a JavaScript file from within an extension, as the eZ publish **Online Editor** does.

{append-block} is similar to {set-block}, but instead of setting (replacing) the content of a variable, you append content to an existing variable. This is used in the Online Editor product, which appends instructions for including the JavaScript files in the <head> section for the extra_header_data variable. The syntax is the same as with {set-block}, but without the name parameter:

```
{append-block scope=<scope> variable=<myvariable>}
{* things you want to assign to *}
{/append-block}
```

Both {set-block} and {append-block} can be used to modify variables outside the current namespace.

Variable Types

Both atomic (simple) and compound variable types can be used in the template language of eZ publish.

Simple Types

The common simple variable types supported by eZ publish are numeric values (integers and floats), strings, and boolean values (although booleans are special functions).

Compound Types

The compound types supported by eZ publish are arrays and objects. Arrays also come in two flavors: **classical arrays** with a numerical index, and **associative arrays**.

Objects are only generated by the kernel functions and operators of eZ publish. Arrays can be created in templates with the `array` and `hash` functions, much like in PHP.

Type Creators

Some types can only be created from PHP, by using operators. Arrays are created with the `array` or the `hash` operator and booleans with the `true()` and `false()` operators. Objects cannot be created in templates directly, but for object-like structures, you can use nested (multidimensional) arrays. Note that like the general objects, there are also methods involved, not only static data.

true()

Creates a true boolean. Remember to use brackets:

```
{true()}
```

false()

Creates a false boolean. Remember to use brackets:

```
{false()}
```

array()

Creates an array.

```
{array(1,2,'tree')} /*creates a mixed type array consisting of
numbers and a string*/
{array(true(),true(),false())} /*creates an array of boolean values*/
```

Like PHP, array elements can be mixed types with numbers, literals, other arrays, and objects.

hash()

Creates an associative array, with the keys being the *odd* parameters and the values being the *even* parameters.

```
{hash(haircolor, 'none, bald', age, 41)}
```

Accessing Elements of Arrays and Objects

Elements of arrays and objects are accessed by the '.' operator. For example, a member called *member* of an object called *object* is accessed as:

```
$object.member
```

Array elements are accessed in the same way:

```
$array.0 {*selects the first element in an array*}
$array.mykey {*selects the element of a hash array with key "mykey"
*}
```

Sections in Templates and their Effects on Variables

Sections are not just restricted to group nodes and node-trees; they are also used in the template language for a variety of constructs. They are used for loops and constructs like if-then-else, and also appear subtly when defining and using variables, even in constructs that do not bear the section name.

This is all due to the use of variable namespaces inherently created by using `section`, `let`, and other constructs for which you can assign a name parameter.

Variable Namespaces

In general, every variable in an eZ publish template is associated with a namespace. By introducing sections for loops and other constructs, a variable defined one of these sections is assigned to the namespace associated with that section. Failure to understand this is one of the main causes of errors in template programming.

Predefined Variables

Most templates have one or more template variables available for use within them. Some are passed from a template higher in the template stack; others are set by the eZ publish kernel modules and functions.

The following sections list the most common and useful variables, but plenty more exist, and even more will be present in future releases.

Effects of Caching and Availability of Predefined Variables

Caching has a major influence on the available predefined variables. As a rule of thumb, it is best to always think of the situation with view caching 'on'. The main variables used are `$module_result` (in the page layout templates) and `$node` (in the view templates).

Variables Available in the Page Layout Templates

`$module_result` is the basic variable available in page layout templates. For instance, `$module_result.content` contains the result of module/function processing and template compiling/processing in the form of XHTML output.

`$module_result`

Attribute	Type (in template)	Comments
<code>content</code>	string	Content generated by the requested module/function
<code>node_id</code>	string	Current node ID
<code>content_info</code>	object	Compound object containing useful data

`$module_result.content_info`

Attribute	Type	Comments
<code>site_design</code>	string	Current site design used.
<code>node_id</code>	string	Current node ID.
<code>parent_node_id</code>	string	Parent node ID of the current node.
<code>node_depth</code>	string	Current node depth.
<code>url_alias</code>	string	Current URL alias for use in nice URLs.
<code>object_id</code>	string	Object ID (a numerical value).
<code>class_id</code>	string	Class ID (a numerical value).
<code>section_id</code>	string	Current section ID (a numerical value)
<code>navigation_part_identifier</code>	string	Current navigation part (used in admin templates).
<code>viewmode</code>	string	Current viewmode: very useful for determining navigation elements.

Attribute	Type	Comments
language	string	Current language.
view_parameters	array	Additional view parameters. Since version 3.3, this array contains the offset parameter for the Google or plain navigators and year, month, day parameters for the monthview navigator. These navigators are templates you can include in your own template (see the files in design/standard/templates/navigators).
role_list	array	Array of role objects for the current user.
role_id_list	array	Array of the role IDs.

The navigators mentioned in the `view_parameters` entry add navigation elements in the style of Previous and Next to your pages, much like the elements used to move between pages on Google search results. The monthview navigator displays links to objects published on certain dates.

Variables Available in Content Views

The main variables set in the view templates are `$node` and `$view_parameters`. The former is an entire object, with a wealth of data ranging from generic node attributes, such as creator, creation date, depth in the node tree, to others that give access to attributes, related objects, and more. On the attribute level, along with the available content, depending on the datatype, many object methods and specifiers offer additional output formats (for example, XML text fields converted to XHTML and image variations).

`$node`

Attribute	Type	Comments
node_id	string	Current node ID
parent_node_id	string	Parent node ID
main_node_id	string	Main node ID (an object can have multiple locations or nodes)

Attribute	Type	Comments
contentobject_id	string	Content object ID, necessary for editing
contentobject_version	string	Current version of the object
contentobject_is_published	string	Can be either 1 (published) or 0 (not published)
depth	string	Depth in the node tree, 1 is the root node
name	string	Name of the object, as defined in the class it is derived from
path_identification_string	string	Part of the URL
data_map	array	Array of ezcontentobjectattribute objects
object	object	Class: ezcontentobject
parent_object	object	Class: ezcontentobject of the parent node
sort_array	array	Sort method, as defined in the admin interface
path_array	array	Array with the node IDs starting from 1, the root node and so on
subtree	array	Entire subtree of this node
children	array	Array of child objects at the level immediately below this node
contentobject_version_object	object	Current version of the object as a version object
creator	object	User object of the creator: you can use <code>\$node.creator.name</code> for displaying the owner of the object

\$node.object

The `$node.object` variable exposes the content object assigned to the current node.

Attribute	Type	Comments
<code>section_id</code>	string	Current section ID
<code>owner_id</code>	string	Owner ID
<code>contentclass_id</code>	string	Class ID of the object
<code>class_name</code>	string	Class name of the object
<code>name</code>	string	Name of the object; should be the same as <code>\$node.name</code>
<code>published</code>	string	datetime value of the creation time as UNIX timestamp
<code>modified</code>	string	datetime value of the last modified time as a UNIX timestamp
<code>can_read , can_create, can_edit, can_translate, can_remove</code>	boolean	Show what the current user can do with this object and whether the current user can create children of the current object
<code>can_create_class_list</code>	array	Array of class IDs, for which the current user can create child objects
<code>related_contentobject_array</code>	array	Array of related objects, as defined in the admin interface
<code>related_contentobject_count</code>	string	Number of related objects, as defined in the admin interface

View Parameters

In some cases, view parameters like `offset` and `limit` are passed to templates. Mostly, this is done from a template file higher in the template stack. These are accessible by the `$view_parameters.offset` and `$viewparameters.limit` variables.

Variables Available in Edit Templates

In edit templates, the main variables are `$class` and `$object`. From these (compound) variables, the relevant items such as class attributes, (initial) node placement, versions, and related objects are obtained for editing objects.

It is perhaps challenging at first sight, but you should take a look at the edit templates in `design/standard/content` to see how edit templates are set up. You can override these edit templates to incorporate them in your site design.

Other Mechanisms

One of the mechanisms to have variables is to use the `ezpreference` system. You can set user-bound preference variables with `/user/preferences/set/<variable name>/<value>`. The variables can be accessed by `ezpreference('<variable name>')`.

Be careful when using the preference system, as it is not part of the content view cache mechanism. You will have to resort to disabling the view caching with `{set-block scope=root variable=cache_ttl}0{/set-block}` and use cache blocks to optimize the response times.

Using Variables across Templates

This topic is currently not addressed in eZ publish, in spite of its obvious benefits. From version 3.4, a mechanism will be introduced where variables can be passed by adding them to the URL in a special way. The URL-passed variables are then also part of the default view cache keys.

The mechanism is as follows: you simply append a variable enclosed in brackets followed by a slash and the value of the variable. For example:

```
http://mysite.com/content/view/full/2/(myvariable)/myvalue
```

This assigns the value `myvalue` to the `myvariable` variable. Its value can be called as a child of the predefined `$viewparameters` template variable. So:

```
{ $view_parameters.myvariable }
```

This will output the `myvalue` string.

You can also override normal view parameters, such as `offset` and `limit`, by using these as your custom template variables.

In version 3.3, the best method is to use the `ezpreference` mechanism.

If your parameter values contain spaces, they will be converted to a URL encoded form by most browsers. Therefore, you should always decode these values before using them. The `urldecode` operator is not specified by default in 3.3, but you can easily add this operator through the `template.ini` file, which contains the `urlencode` operator as a direct match to the PHP equivalent.

Controlling Template Output Flow

Here we will look at some of the basic language constructs that allow us to control the flow of template output:

- The `{section}` construct
- Looping with `{section}`
- `{switch}` constructs

At the end of the section, we will revisit namespaces and see how to better handle the problem of a variable definition being tied to the section in which it is defined.

Section

`{section}` is one of the most important functions in the template engine. It provides looping over arrays and numeric ranges (equivalent to a `for` statement), and conditional control of blocks and sequences (if-then-else). It is controlled by a series of input parameter and sub-functions.

An example of its use for looping is:

```
{let myarray=array(1,2,3)}
{section name=myloop loop=$myarray}
    {$myloop:item}
    {delimiter},&nbsp;{/delimiter}
{/section}
{/let}
```

This will output the string "1, 2, 3".

In general the syntax for `{section}` is:

```
{section name=<myname>
    show=<showcondition>
    loop=<array>
    sequence=<sequence-array>
    max=<max iterations>
    offset=<offset in the index for loops>}
/* code if showcondition is true */
{delimiter}
    --- /* this is printed/executed except for the
        last iteration */
{/delimiter}
{section-else}
    /*code if showcondition is false*/
{/section}
```

Parameters

Having seen the basic syntax of the `{section}` function, let's look more closely at each of its parameters.

name

`name` defines the namespace for all generated template variables; see `loop` and `sequence` for a list of generated variables.

loop

`loop` defines the data that the section should loop over. For every iteration of the loop, it sets the template variable `$name:item`. The data can be an array (array of scalars or an array of objects), in which case each item in the array is traversed sequentially.

The data can also be a number that determines the number of iterations (a negative number makes the iteration go backwards).

It's possible to constrain the number of iterations performed and control the iteration of single elements. See parameters `max` and `offset` and sub-children `{section-exclude}` and `{section-include}`.

Each time the section iterates, it sets four template variables in the new namespace. The variables are `index`, `number`, `key`, and `item`.

- `index`: A number that starts at 0 and increases for each iteration.
- `number`: Behaves in the same way as `index` but starts at 1.
- `key`: If an array is being iterated, `key` holds the key of the current item. Otherwise, the current iteration index is set.
- `item`: If an array is being iterated, `item` holds the value of the current item. Otherwise, the current iteration index is set.

show

This parameter determines whether the section block should be shown or not. If the parameter is not present or is true (either a boolean true, non-empty array, or non-zero value) the `{section}` block is shown, otherwise the `{section-else}` block is shown. This is quite useful for conditional inclusion of template code depending on a variable.

When the `{section-else}` block is used, no looping occurs.

sequence

Defines a sequence that is iterated as the normal `loop` parameter. However, the difference is that the sequence will wrap and only support arrays. The current item will be set in the sequence template variable. This parameter is useful if you want to create, for instance, alternating colors in lists.

max

Determines the maximum number of iterations. The value must be an integer or an array; if it's an array, the number of elements of the array is used.

offset

Determines the start of the loop array for the iterations. The value must be an integer or an array; if it's an array, the number of elements of the array is used.

section-else

Determines the start of the alternative block that is shown when show is false.

delimiter

{delimiter} determines a block of template elements that should be placed between two iterations.

section-exclude and section-include

{section-exclude} and {section-include} add filter rules for excluding or including a loop item. The rules will be run one after another, as they are found. The rule will read the match parameter, which expects a boolean value, and changes the current accept/reject state for the current item. The default action is to accept all items. The match parameter can match any template variable available, including loop iterators, keys, and items, but not the loop sequence itself.

Sequences and iteration counts (number and index) will not be advanced if the loop item is discarded.

For example:

```
{section name=Myloop loop=array(a,b,c,d,e)}
{section-exclude match=true()}
{section-include match=array(c,d)|contains($Myloop:item)}
{$Myloop:item}[key:{$Myloop:key}][number:{$Myloop:number}]{delimiter}
{/delimiter}
{/section}
```

This will produce:

```
c[key:2][number:1],d[key:3][number:2]
```

Note that the \$Myloop:number variable only increments for each iteration while the variable \$Myloop:key is still the correct array key (which starts at 0 for ordinary arrays).

If-then-else Constructs with Section

If-then-else constructs are carried out using the show parameter. This parameter expects a boolean or the equivalent of a boolean (a zero or non-zero value). A non-empty array will evaluate to true. For example:

```
{let a=array(1,2,3,4)}
<p>-if-then-else</p>
{section name=if show=$a}
{*this loop will be selected because an array will evaluate to true
if it has one ore more elements*}
{section loop=$a}
  a={$:item}
{/section}
{section-else}
  <p>nothing to show</p>
{/section}
{/let}
```

The output will show:

```
if-then-else
a=1 a=2 a=3 a=4
```

Loops with Section

The following examples demonstrate how to use loops:

A Fairly Normal Loop using a Sequence Array

```
{section name=myloop loop=array(1,2,3,4,5)
sequence=array('odd','even')}
  {delimiter} --- {/delimiter} <br/>
  {$myloop:item} - {$myloop:sequence} <br/>
{/section}
```

This will output:

```
1 - odd
---
2 - even
---
3 - odd
---
4 - even
---
5 - odd
```

A Negative Loop:

```
<p> A negative loop: <br />
{section loop=-3}
  {$:item} &amp;
{/section}
</p>
```

This will output:

-1 & -2 & -3 &

A Positive Loop with Limits and an Offset:

```
<p> A positive loop with offset
{section loop=8 max=4 offset=2}
{${item}} {delimiter} &amp; {/delimiter}
{/section}
</p>
```

This will output:

3 & 4 & 5 & 6

You may also use {run-once} blocks inside loops to make sure parts are only processed once for all iterations. A trivial example:

```
{section loop=array(1,2,3,4)}
{run-once}
<p>Here come 4 iterations of a loop</p>
{/run-once}
{* here a nameless loop is used, so the name is omitted
but not the colons *}
<p>{${item}}</p>
{/section}
```

This will output:

```
<p>Here come 4 iterations of a loop</p>
<p>1</p>
<p>2</p>
<p>3</p>
<p>4</p>
```

Switch Constructs

The {switch} function allows conditional control of output and is an alternative for if-then-else constructs, when many possibilities need to be coped with. For instance, you can display some XHTML code, depending on a template variable. The matching can be directly between two types or for an element in an array.

The matching is done by creating one or more {case} blocks inside the {switch} block.

For eZ publish versions up to 3.3, there must always be one default case present. A default case is created by inserting a {case} block without any match parameters. From version 3.4 onwards, the default case is not required.

The parameter to a case can either be match, which determines the value to match against, or in, which must contain an array. match does a direct match, while in looks for a match among the elements in the array. The in parameter behaves differently if the key

parameter is used, which must be an identifier. It then assumes that the array sent to `in` has an array for each element and uses the key to match a key in the sub-array. Take a look at the following examples:

Simple Scalar Matching

```
{let matchme='Skien'}
{switch name=city match=$matchme}
  {case match='Paris'}
    This does not match
  {/case}
  {case match='Skien'}
    This matches, lets visit the ez crew
  {/case}
  {case}
    default, must be present!!
  {/case}
{/switch}
{/let}
```

Matching with an Array

```
{let matchme=4}
{switch name=city match=$matchme}
  {case in=array(1,2,3,4,5)}
    This matches, you gained 4 points
  {/case}
  {case}
    {* default, must be present for versions up to 3.3!!*}
  {/case}
{/switch}
{/let}
```

Variable Namespaces Revisited

Now that you have learned about the various section constructs, it's time to tackle the effects on variable namespaces more deeply.

In general, a variable definition is tied to the section in which it is defined. At the top level, variables are associated with the implicit root namespace of a template. When defining variables inside sections, they are associated with the depth level of their section by the namespace built using the section names, separated by a colon (:).

For example:

```
{* start of template *}
{let a=2
  b=array(1,2,3)
}
{* variable $a is now in the root namespace *}
{section name=level1 loop=$b}
  {let a=$level1:item}
```

```

        <p> a inside loop is {$level1:a} </p>
        <p> root level variable a is {$a} </p>
    }/let}
}/section}
}/let}

```

This will output the following:

```

a inside loop is 1
root level variable a is 2
a inside loop is 2
root level variable a is 2
a inside loop is 3
root level variable a is 2

```

Sections without a Name Parameter

In the absence of a name parameter, you access the current namespace with `$.`. When using a named `{let name=myname}` construct surrounding a nameless section, the namespace will actually inherit the namespace from the `{let}` construct.

Set or Change a Variable from an Outer Namespace

You cannot use `{set}` for changing variables outside a loop, but the `{set-block}` can be used to achieve the desired results. For example:

```

{let a=3
    b=array(1,2,3,4,5)}
{section name=test loop=$b}
{let a=$test:item}
<p>local a={$test:a}, global a={$a}</p>
{set-block scope=root variable=a}{$test:a}{/set-block}
<p>again after set-block, local a={$test:a}, global a={$a}</p>
}/let}
}/section}
}/let}

```

This will produce the following output:

```

local a=1, global a=3
again after set-block, local a=1, global a=1
local a=2, global a=1
again after set-block, local a=2, global a=2
local a=3, global a=2
again after set-block, local a=3, global a=3
local a=4, global a=3
again after set-block, local a=4, global a=4
local a=5, global a=4
again after set-block, local a=5, global a=5

```


Avoiding Namespace Problems

Since version 3.3, you can deal with (or avoid) namespace problems by introducing the `var` parameter in sections. `var` specifies a variable name that is valid inside the loop and every sub-level of it.

For example:

```
{section var=outer loop=array(a,b,c)}
  {section var=myloop loop=array(1,2,3,4,5)}
    {$myloop.item}{$outer.item}{delimiter}-{/delimiter}
  {/section}
{delimiter}<br />{/delimiter}
{/section}
```

This will result in:

```
a:1 - a:2 - a:3 - a:4 - a:5
b:1 - b:2 - b:3 - b:4 - b:5
c:1 - c:2 - c:3 - c:4 - c:5
```

From eZ publish version 3.4 onwards, you can also use the `var` parameter instead of a `name` parameter in `{switch}` constructs to determine the iterator.

Using Functions from Kernel Modules

The eZ publish kernel modules can expose some of their functions for use in templates. This is done by the standard module file `ezfunctiondefinition.php`, which you will find in various modules such as `content`, `user`, `collaboration`, and so on.

The general syntax is as follows:

```
fetch(module, function, parameters)
```

Functions in the Content Module

We will now discuss the most common functions from the `content` and `user` modules. The following table provides a summary of the available functions. Some of the important functions are treated in more detail.

Function	Description
<code>node</code>	Fetches a single node based on the node ID.
<code>object</code>	Fetches a single object based on the object ID.
<code>list</code>	Fetches all or filtered subset of child nodes with a certain depth (default depth = 1).

Function	Description
<code>tree</code>	The same as lists, but with unlimited depth.
<code>list_count</code>	Gives the number of child nodes for a given node with a certain depth (default depth = 1).
<code>tree_count</code>	Gives the total number of child nodes for a given node.
<code>search</code>	Searches for objects with specified criteria.
<code>locale_list</code>	Returns the locale objects defined for the current siteaccess.
<code>translation_list</code>	Returns the translations defined for the current siteaccess.
<code>non_translation_list</code>	Returns the list of possible translations for a certain object, excluding those that already exist.
<code>class</code>	Returns the class object structure for a given class ID.
<code>class_attribute_list</code>	Returns the class attributes as an array of attribute objects for a given class ID (and optional version).
<code>class_attribute</code>	Returns a single class attribute structure with a given attribute ID and optional version.
<code>trash_count</code>	Returns the number of objects in the trash.
<code>trash_object_list</code>	Returns the objects in the trash.
<code>draft_count</code>	Returns the number of drafts for the current user.
<code>draft_version_list</code>	Returns the objects in draft for the current user.
<code>pending_count</code>	Returns the number of pending objects for the current user (objects that are not yet published because they are subject to a pre-publishing workflow).
<code>pending_list</code>	Returns the array of pending objects for the current user.

Function	Description
<code>version_count</code>	Returns the number of versions for a given content object.
<code>version_list</code>	Returns the versions for a given object (with optional limit and offset).
<code>can_instantiate_class_list</code>	Returns the classes for which the current user can create objects with optional parent node ID and class group ID.
<code>can_instantiate_classes</code>	Same as above, but only for a optional given parent node ID.
<code>contentobject_attributes</code>	Returns the attributes for a given version object with optional language code.
<code>bookmarks</code>	Returns the list of bookmarks for the current user.
<code>recent</code>	Returns the list of recent items for the current user.
<code>section_list</code>	Returns the list of sections IDs defined.
<code>tipafriend_top_list</code>	Fetches a list with node objects based on the most tipped objects. Offset and limit may be supplied.
<code>view_top_list</code>	Fetches a list of most viewed node objects. Needs the log file counter script to be run.
<code>collected_info_count</code>	For the information collector system; fetches the number of collected items for a given object if <code>object_id</code> is sent, or a specific attribute if <code>object_attribute_id</code> and value is sent. This can for instance be used to display the results of a poll (attribute with <code>ezoption</code>) or the number of feedbacks from a form.
<code>collected_info_count_list</code>	Similar to <code>collected_info_count</code> but counts a given attribute (by its ID) and returns a list of counts for each element in that attribute. This assumes that the attribute is of type <code>ezoption</code> . Again, this is useful for polls.

Function	Description
<code>collected_info_collection</code>	Fetches the collection with a given ID if <code>collection_id</code> is specified. If the <code>contentobject_id</code> parameter is used instead, it will find the first collection item that is related to the current user and object ID. This works for both anonymous and registered users. This can, for instance, be used to fetch a user's response to a given form object. If there are multiple collections, the first (may be randomly selected) is chosen.
<code>object_count_by_user_id</code>	Returns the number of objects created by a given user ID and class ID. Useful for statistics.
<code>same_classattribute_node</code>	Returns a list of nodes where the value for a given attribute ID and datatype name are the same. This is faster than a search function.
<code>keyword</code>	Returns the keywords and nodes that have keywords matching a certain portion defined in the <code>alphabet</code> parameter. Useful to create keyword indexes and is used by the PDF export system. Optional parameters are a limit and offset for browse interfaces.
<code>keyword_count</code>	Returns the number of keywords for a given class and 'alphabet string'.

Fetching a Single Node or Object

To fetch a single node data structure, you need to provide the `<node id>` as an integer value:

```
fetch(content, node, hash(node_id, <node id>))
```

For example, to get the current node in a page layout template, you can use the following code fragment.

```
{let thisnode=fetch(content, node, hash(node_id,
$module_result.node_id))}
{...
{$thisnode.name} {* print out the node name *}
{...
{/let}
```

If you have only an object ID available, you can fetch it with this command:

```
fetch(content, object, hash(object_id, <object id>))
```

In general, the use of fetching objects by object ID is not necessary as the object data structures are available directly.

Fetching Node Lists and Node Trees

The syntax is `fetch(content, list|tree, hash(<parameters>))`.

The difference between the `list` and `tree` functions is only the default `depth` parameter. For `list`, the depth is by default set to 1; with `tree` the default depth is set to 0 (meaning unlimited depth). All other parameters are shared by both functions.

Parameters are specified by an associative array. Some parameter values also be arrays.

Parameter name	Type	Required	Default	Description
<code>parent_node_id</code>	integer	yes	none	The node ID for which you want to fetch the children.
<code>sort_by</code>	array	no	node name	The sort method you want to apply.
<code>offset</code>	integer	no	<code>false()</code>	The subnode to start from, mainly used for navigation lists.
<code>limit</code>	integer	no	<code>false()</code>	The maximum number of nodes to return.
<code>depth</code>	integer	no	1 or <code>false()</code>	The depth of subnodes to recurse. For lists, this is preset at 1, for tree, this is preset at false (unlimited depth). 0 also means unlimited depth.
<code>depth_operator</code>	string	no	le	The operator to use for determining how to treat the depth value. By default this is set to <code>le</code> but you can also use <code>eq</code> to specify the nodes at a certain depth level only. Other operators are not yet implemented.

Parameter name	Type	Required	Default	Description
<code>class_id</code>	integer	no	<code>false()</code>	Limit the subnodes to the given class (single).
<code>attribute_filter</code>	array	no	<code>false()</code>	Select nodes returned based on attribute values.
<code>extended_attribute_filter</code>	array	no	<code>false()</code>	The extended attribute filter can call filter functions from an extension. Currently, no such extensions are in the base distribution.
<code>class_filter_type</code>	string	no	<code>false()</code>	Type of filter: either include or exclude.
<code>class_filter_array</code>	array	no	<code>false()</code>	Array of class identifiers (names) or class IDs (integers).
<code>group_by</code>	array	no	<code>false()</code>	Group nodes together (currently only for date parts of published and modified object attributes).
<code>main_node_only</code>	boolean	no	<code>true()</code>	When an object has more than one location, this flag ensures that only the main node is used (avoids duplicates in for example 'what's new' lists).

Some of the parameters are discussed here in more detail.

Filtering

Filtering can be done based on (an array of) class IDs, and also attributes. The syntax for class filtering is based on two parameters: the `class_filter_type`, which can be either include or exclude, and the array of numeric class IDs or class identifiers. For example:

```
{let mylist=fetch(content,tree,hash(...,class_filter_type,
'exclude',class_filter, array(12, 14))}
```

For portability of templates, you can use class identifiers as follows:

```
{let mylist=fetch(content,tree,hash(...,class_filter_type,
'exclude',class_filter, array('book','magazine'))}
```

The general syntax for attribute filtering is an array of arrays that are combined with the keyword `and` or `or`. The array elements are the attribute ID (as found in the class attribute list in the administrator interface), an operator, and a value. For example:

```
{let mylist=fetch(content,list,hash(..., attribute_filter,
                                array('or',array('myclass/color','=','blue'),
                                array('myclass/width','>','20'))}
```

Instead of using the numeric attribute ID, you can also specify a combination of class and attribute identifiers such as 'book/title', which enhances portability of templates. The usual operators can be specified: `>`, `<`, `>=`, `<=`, `=`, and `!=`.

Sorting (with `sort_by`)

Sorting can be done either on one of the basic object attributes or on the value of an attribute. You can also combine sorting methods in an array, for example, to sort by class and then sort by name. With a boolean, you can specify whether a sort rule should be ascending or descending.

The syntax for sorting on the basic object attributes is as follows:

```
{let mylist=fetch(content,list, hash(..., sort_by, array(
                                array('class',true()), array('name',false()))}
```

Counting the Objects of Certain (or All Types)

Counting nodes can either be done with the `tree_count` or `list_count` functions. The difference between them is only the default depth parameter, which is '1' for `list_count`. Both functions expect the following parameters:

Parameter name	Type	Required	Default value	Description
parent_node_id	integer	yes	none	The node tree starting from this node ID
depth	integer	no	1 or 0	Depth 0 for <code>tree_count</code> , 1 for <code>list_count</code>
class_filter_type	string	no	false()	Either 'include' or 'exclude'
class_filter_array	array	no	false()	An array with class IDs or class identifiers

Parameter name	Type	Required	Default value	Description
attribute_filter	array	no	false()	Array of attribute filters as in list and tree functions
main_node_only	boolean	no	true()	Counts only the main node placements in case of objects with multiple placements

The syntax for the parameters is the same as discussed for the tree and list functions.

Displaying Version Information

To display version information, you can use either the supplied `version_list` function (best used in page layout templates) or from the `$node.object` structure (best used in view templates to minimize the processing overhead).

The syntax for the `version_list()` function is:

```
{let versions=fetch(content,version_list, hash(contentobject,
$object, limit, $limit, offset, $offset))}
```

Parameter name	Type	Required	Default value	Description
contentobject	object	yes	false()	The object for which you want to list the versions
limit	integer	no	false()	The number of versions to fetch
offset	integer	no	false()	An offset in the list of versions

The `version_list` function returns a list of version objects. A method for showing audit trails with the `$node.object` structure is discussed in the tips and tricks section.

Fetching the Current User

The current logged-in user and its associated object can be obtained by using the `current_user` function from the user module. If there is no logged-in user, the anonymous user is returned. The user object has the basic parameters from any object,

and user-specific attributes set from the `ezuser` datatype (user ID, login ID, and e-mail). If you use the default user class, the other attributes are `name` and `firstname`.

The object structure returned also contains user-specific data on the roles that are assigned and the user groups the user belongs to.

Others

Listing all the functions here would be reaching too far, but some functions to consider using are the bookmark lists (which can be used as user-specific menus), pending lists and collaboration items (for workflows), and so on. You should consult the online documentation and the user forum for more information.

Increasing Performance with Caching

Besides the overall caching mechanisms (view caching, INI caching, template compilation), you can also manipulate the caching of parts of your templates by using customized cache blocks.

Overall Caching

The main cache use is controlled by INI variables in the template and content sections of the `site.ini.append.php` file. During template development, you may want to disable view caching and template caching (compiling), so that your modifications show up immediately without resorting to caching.

For small edits to your templates, you can also consider using the admin interface. When you change templates and save them, the relevant portions of the cache are selectively cleared.

Cache Blocks

Cache blocks store the result of dynamic template code contained in these blocks in a plain text file. The next time the same code is requested, the plain text file is loaded without template processing overhead (but the template blocks also introduce processing overhead that can grow significantly if you employ many cache blocks).

The `{cache-block}` construct normally needs two parameters: the cache block keys, which control the scope of the cache block, and an expiry period.

Whether or not you provide keys, the template cache system will create additional ones based on the template file name and path, the position (!) of the cache block in the file and the current siteaccess. So, multiple cache-blocks in a template file with no explicit

keys still get a unique key. This works even for different site accesses that use the same design (and templates within), in which the content differs with site access.

The use of keys is mandatory when you use the cache block in a template and the content of the cache block can be different. For view templates, I would recommend at least the node ID and the user ID (if different users have different roles). You may add other variables here too.

Alternatively, when you have a large number of users who share a few roles, you may be better off with using the `role_id_list` of the `$current_user` object for cache block keys in page layout templates.

View templates employing cache blocks can be necessary when you provide more than the basic attribute content, provided you first switch off the view cache at the top of the template with the directive `{set-block scope=root variable=cache_ttl}0{/set-block}`

For use in page layouts, and when you have navigation menus which remain constant, it is safe to use no keys at all for the surrounding cache block. Otherwise, you will need a key which discriminates the different content (for example, the user ID when roles affect the menus to display).

Keys

The `keys` parameter uniquely identifies a cache block. The default keys used by eZ publish are the template name and block position. You can specify a key either as a single variable or as an array. This means that when you use, for example, the array of role IDs as key, each type of user defined by assigned roles will cause a new cache block to be created.

For example, in a page layout template you may want to surround a `treemenu` block with:

```
{cache-block keys=$module_result.node_id}
{*template code*}
{/cache-block}
```

Expiry

If you don't specify the `expiry` parameter, eZ publish will automatically expire the cache block in two hours or if any content is published. This means that if you publish anything on your site all cache blocks will be expired. However, you can disable content expiry for certain blocks with the `ignore_content_expiry` parameter:

```
{cache-block ignore_content_expiry}
{* Always cached template output, regardless of the new objects being
published*}
{/cache-block}
```

If the default expiry does not fit your needs, you can specify the expiry time manually, in seconds.

The following example expires a certain cache block every five minutes:

```
{cache-block expiry=300}
{* comment list *}
{/cache-block}
```

Custom Template Operators

As eZ publish is an extensible CMS, you can create your own template operators and functions. An easy way to add more operators is to use (standard) PHP functions that take a single argument; you can add their definitions to the `template.ini` file.

For example, the following settings are commonly added for the author's eZ publish portals in `settings/override/template.ini.append`:

```
[PHP]
PHPOperatorList[lefttrim]=ltrim
PHPOperatorList[urldecode]=urldecode
PHPOperatorList[striptags]=strip_tags
```

This makes available the new template operators `lefttrim` (like the `trim` function, but operates only on the start of a string), `urldecode` (reverse of the `urlencode` operator, which is added as a PHP operator in the default `template.ini`) and `striptags` (which strips XML and HTML tags of a given string). When you edit your `template.ini` file, remember to clear the INI cache in the `admin->setup->cache` interface.

The Template Override System

The template system itself can be configured to provide a dedicated mechanism of choosing *which* templates are used. This is accomplished by the specification of design directories and their specific content. For example, when you place a `pagelayout.tpl` file in `design/<yourdesign>/templates`, it will take precedence over the `pagelayout.tpl` found in `/design/standard/templates`.

Through the mechanism provided by the override system (with the `override.ini.append` file in your site access settings directory), the flexibility is virtually endless and can be controlled down to a single node based on one or more conditions, such as the class of the object, a certain class of an XML tag, the depth of a node in the node tree, and more.

Using Cascading Effects in Templates

The template system will first start by looking for a given template file in the `override.ini` file. However, if nothing is defined there, the search logic is implemented to first look in the `design/<your design>/templates` directory structure. If nothing is available there, it will go through the additional design directories specified in `site.ini.append`. Finally, it will fall back to the default design directory, which should be standard.

To know which template should go where, you need to study the `design/standard/template` directory structure.

Overriding Templates Using Specific Conditions

The `override.ini` file contains entries that are matched from top to bottom. The first match found is used. This is also why you can specify a priority in the template admin interface. When editing the file manually (which you will want to do, since the Admin interface does not provide all the possible features), be careful about what you put where.

It is important that you put the most stringent specifications at the top, because if a more general match is encountered, it will be used. This is an error made by a lot of novices, who wonder why their template is not used.

Syntax of `override.ini.append.php`

The `override.ini.append.php` specifies blocks identified by a 'header' surrounded by square brackets. You can put anything in this header, as its sole function is to separate a certain block from other blocks. However, best is to use a meaningful identifier. When editing the templates through the admin interface, the header is simply the filename of your template stripped from the trailing `.tpl`. You may also use this convention as it makes sense provided you use meaningful filenames. The example below is from a hypothetical eZ publish site and design:

```
<?php /* #?ini charset="iso-8859-1"?
[calendar_folder]
Source=node/view/full.tpl
MatchFile=calendar_folder.tpl
Subdir=templates
Match[class]=1
Match[node]=65

[calendar_event_line]
Source=node/view/line.tpl
MatchFile=calendar_event_line.tpl
Subdir=templates
Match[class]=28
*/ ?>
```

[calendar_folder] and [calendar_event_line] are the headings denoting the start of a new block. The parameters Source, MatchFile, SubDir, and Match[condition] are defined below.

Source

Source is the template that will be called by default, if you do not provide your own template file.

SubDir

SubDir is the directory inside your design/<yourdesign>/override directory, which holds the template overrides.

MatchFile

MatchFile specifies the filename of your template that will override the Source file relative to the SubDir directory.

Match

Match can actually consist of more than one Match condition; specify them with one entry for each condition to match.

The conditions are listed in the following table:

Condition	Value type	Description
class	numeric, class ID	Matches one (or more) of your object classes
node	numeric, node ID	A specific node
depth	numeric, node depth	The depth of the node in the node tree
section	numeric, section ID	A specific section
object	numeric, object ID	A specific object regardless of its node location
navigation_part_identifier	navigation part identifier	Used in the admin interface to determine which navigation part it belongs to (content, setup, etc.)

Condition	Value type	Description
parent_node	numeric, node ID	Matches the node ID of the parent
url_alias	string	Matches the URL alias of a node
class_identifier	string	can be used instead of class (from version 3.4 onwards)

Common Template Tasks

In this section, we will look at some common template tasks and how to solve them. We will look at:

- Navigation menus
- Adding edit functions to your pages
- Date and time tasks
- String and text manipulation
- A custom user experience
- Further miscellaneous tasks

Navigation Menus

Providing suitable menus for users to navigate your site is a must to make your site both usable and accessible. In this section, we will look at how to create three of the most common types of navigation menus.

Top Level Menu

Let's create a top-level menu by fetching the top-level elements just below the root node, using the `treemenu` operator. It is to be used inside a `pagelayout.tpl` file.

In the following code, the top-level nodes are all of class `folder` with class ID one. The first parameter to the `treemenu` operator, `module_result.path`, contains the required array of node IDs from the top-level node to the current node. The top-level node, specified in the second parameter, is set to 2 (the default node of the root-level node for the content tree). The next parameter is the class of the nodes (`folder`). Then comes `depthskip`, which is set to 1, and the final parameter indicates the depth from there (zero, because we don't want to go any deeper than the level immediately below the root node).

```

{let topMenus=treemenu($module_result.path,2,array('folder'), 1, 0)}
{section name=topMenu loop=$topMenus}
{section show=$topMenu:item.is_selected}
<a href={$topMenu:item.url_alias|ezurl}
  title="{ $topMenu:item.text}"
  class="topmenuselected">{$topMenu:item.text}</a>
{section-else}
<a href={$topMenu:item.url_alias|ezurl}
  title="{ $topMenu:item.text}">{$topMenu:item.text}</a>
{/section}
{/section}
{/let}

```

In this example, the `class` attribute is used to define an appropriate CSS highlight for the current part of the node tree.

Breadcrumb Navigation

In order to create breadcrumb-like navigation structures, the `path` array is used to loop over the nodes from the top level down to the current node. To avoid very long strings, the link display is shortened to 60 characters per path item. A cache block is also used for performance.

```

{cache-block keys=array("path",$module_result.node_id)}
<p class="path">
  HOME &gt;
  {* we'll skip the root node name here by usign an offset of 1 to loop
  over the node ids *}
  {section name=Path loop=$module_result.path offset=1 }
  {section show=$Path:item.url}
  <a href={$Path:item.url_alias|ezurl}>
    {$Path:item.text|shorten(60)|wash}
  </a>
  {section-else}
  {$Path:item.text|shorten(60)|wash}
  {/section}
  {delimiter}
  /
  {/delimiter}
  {/section}
</p>
{/cache-block}

```

Tree Menus

This time we use the efficient `treemenu` operator to create a genuine tree-like menu. In the following example, we start one level below the root node (hence the `depthskip` of 1) and allow for five levels down the node tree.

To indent, we use the depth level with `$.item.level` to change the `margin-left` property, shifting the menu entry ten pixels to the right with every depth level.

```
{cache-block keys=array($module_result.node_id)}
<div style="width: 150px; border: none">
  <div class="menuhead">{$module_result.path.1.text}</div>
  <ul>
    {let mainMenu=treemenu
      ($module_result.path,$module_result.node_id,
       false(),1, 5)}
    {section name=Menu loop=$mainMenu}
      <li style="margin-left: {$:item.level}0px;"
        {section show=$:item.is_selected}class="menuselected"
        {section-else}class="menu"{/section}>
        <a href={$:item.url_alias|ezurl}
          {section show=$:item.is_selected}
            class="menuselected"
          {/section}
          title="{ $Menu:item.text}">
            {$Menu:item.text|shorten(50)}</a>
        </li>
      {/section}
    {/let}
  </ul>
</div>
{/cache-block}
```

Adding Edit Functions to Your Templates

There are a number of times when you may wish to have facilities for users with sufficient privileges to add, edit, or remove content directly from the page where it is displayed. Here is a selection of such situations.

Allow Users to Add Content to Your Site

Adding a button and a menu to create objects of certain classes is defined in the roles settings. This is accessed in the current node variable. In view templates this variable is available by default, but if you want to use this in a `pageLayout.tpl` file, you will have to fetch the `$node` variable explicitly.

```
{section show=count($node.object.can_create_class_list)}
  <form method="post" action="/content/action">
    <input type="hidden"
      name="NodeID"
      value="{ $module_result.node_id}" />
    <select name="ClassID">
      {section name=Classes loop=$node.object.can_create_class_list}
        <option value="{ $Classes:item.id}">
          { $Classes:item.name|wash}</option>
      {/section}
    </select>
    <input class="button" type="submit"
      name="NewButton"
      value="{ 'Create here'|i18n('design/standard/node/view')}" />
  </form>
{/section}
```


Adding an Edit Link

The following code snippet adds an edit link to full-view or line-view templates:

```
{section show=$node.object.can_edit}
  &nbsp;<a href={concat("content/edit/", $node.object.id)|ezurl}>
    [Edit]</a>
{/section}
```

Adding a Remove Button

In order to present users with a button to remove content, provided they have sufficient privileges to do so, a little more work is required. Instead of a simple link, you need to create a small `<form>` structure. In the following code, the existence of a suitable icon in the `<your design>/images/small` directory is assumed. The form structure expects both the node ID and content object ID to be present.

```
{section show=$node.object.can_remove}
<form method="post" action={"content/action"|ezurl}>
  <div style="display: inline;">
    <input type="hidden" name="ContentNodeID"
      value="{ $node.object.main_node_id}" />
    <input type="hidden" name="ContentObjectID"
      value="{ $node.object.id}" />
    <input type="image" name="ActionRemove"
      src={"small/edittrash.gif"|ezimage} />
  </div>
</form>
{/section}
```

Adding a Comment Button

To add a comment button, you can use a small form in which the content class is preset. The only other parameter to provide is the parent node ID.

```
<form method="post" action={"content/action"|ezurl}>
  <input type="hidden" name="NodeID"
    value="{ $node.main_node_id}" />
  <input type="hidden" name="ClassID" value="13" />
  <input class="button" type="submit"
    name="NewButton" value="New comment" />
</form>
```

Date and Time Tasks

To format date and time output, you can use the `%l10n` (with one of the date-related modifiers `date`, `time`, `shortdate`, `datetime`) or the `datetime` operator for more fine-grained control. In general, the `%l10n` operator should be used since it is tied to the locale setting of your site access.

Displaying Tomorrow's Date

To display a date in the future with a fixed offset, you only need to use the localization operator `l10n` and the `sum` and `currentdate` operators. The offset for tomorrow needs to be specified in seconds and added to the current datetime value. A full day is 86,400 seconds, so the code below uses this hard-coded value.

```
<p>Tomorrow: {sum(currentdate(),86400)|l10n( shortdate )}</p>
```

You could further enhance this by creating a small template file that expects a reference datetime expressed in seconds since the epoch, and a number of days.

```
{default ref_date = currentdate()
      number_of_days = 1}
{sum($ref_date,mul( 86400 ,  $number_of_days))|l10n( shortdate )}
```

By using the `default` construct, you are sure some sensible default values are used when this template is called without parameters.

String and Text manipulation

The most basic text-manipulation tasks involve shortening and converting strings, as we shall see in the following subsections.

Limiting Text Output

For limiting text output, you can use the `shorten` operator. For example, to limit node names in line views you can use:

```
{ $node.name | shorten(80) }
```

Limiting XML Text Output

For limiting XML or XHTML text output from the XML text datatype attributes, you cannot simply use the `shorten` operator, due to the presence of the tags. If you implemented the example in the *Custom Template Operators* section earlier in this chapter, you should have the `striptags` operator available for use. The following is an example of a line view of the default link class. The relevant part is highlighted:

```
{default node_name=$node.name}<a
href={ $node.url_alias|ezurl }>{ $node.name|wash}</a> <a
href={ $node.data_map.link.content } style="color: red;">[Direct
Link]</a> ( { $node.object.published|datetime(custom,"%Y-%m-%d
%H:%S" ) } )
<div style="padding-left:20px; padding-bottom:1px; font-size: 80%;">
{ $node.data_map.description.content.output.output_text |striptags|shorten(180) }
</div>
{/default}
```

The XHTML text is first retrieved, has its tags stripped, and is then shortened (operator cascading is from left to right).

Automatic Linking and Conversion

For manipulating plain-text fields (for example, a plain-text forum message), you can add operators for washing, introducing line breaks, and converting links to real XHTML links as follows:

```
{ $forum_text | wash | break | auto_link }
```

Remember that operators are executed from left to right.

Providing a Custom User Experience

You can make certain nodes behave like 'administration' pages by overriding the template with specific (sometimes hard-coded) content. I usually employ a dummy class holding just one attribute—a text string—and then create override templates for the nodes created with this class.

Creating Dummy Nodes

The start of a custom user experience can be created easily with a dummy node called something like My Site below the root node. This page does not relate to content directly. Below this dummy node, you can create other dummy nodes where the content is not set by the content object residing at that node, but by template code inside a dedicated override template. For the example code below, dummy nodes called Preferences, My Groups, and Statistics are created.

A Specific User Panel

Inside a dedicated page layout for a My Site node, the following menu structure can be created. Part of the menu is copied from the default administration interface menu for Personal, but other menu entries point to sub-nodes of My Site. For readability, no CSS styles or `l18n` operators are used.

```
<ul>
<li><a href={"/content/draft/"|ezurl}>My drafts</a></li>
<li><a href={"/notification/settings/"|ezurl}>My notification
settings</a></li>
<li><a href={"/content/bookmark/"|ezurl}>My bookmarks</a></li>
<li><a
href={"/collaboration/view/summary"|ezurl}>Collaboration</a></li>
<li><a href={"/content/pendinglist/"|ezurl}>Pending list</a></li>
<li><a href={"/my_site/preferences/"|ezurl}>Preferences</a></li>
<li><a href={"/my_site/my_groups/"|ezurl}>Groups & Roles</a></li>
<li><a href={"/user/password/"|ezurl}>Change your password</a></li>
</ul>
```

Putting the User Preferences Function to Work

The template code shown here is to be placed inside the override template for the Preferences node. It is imperative that you disable the view cache for this kind of interactive page, otherwise the preference values will not be updated in successive renderings.

The user preference variable set in the example is about showing audit trails. This variable is used in the audit trail example further in this chapter and can be set to ON or OFF.

```
{set-block scope=root variable=cache_ttl}0{/set-block}
<h1>{$node.name}</h1>
<table>
<tr>
<td>Show audit trails</td>
<td>Current:
  {section show=ezpreference('audittrail')}
    <b>{ezpreference('audittrail')}</b>
  {section-else}
    <b>--</b>
  {/section}
</td>
<td>Change to:
  {section show=eq(ezpreference('audittrail'),'on')}
    <a href="{user/preferences/set/audittrail/off"|ezurl}>OFF</a>
  {section-else}
    <a href="{user/preferences/set/audittrail/on"|ezurl}>ON</a>
  {/section}
</td>
</tr>
</table>
```

Showing a User's Groups and Roles

This template code for overriding the view template of the My Groups node below My Site shows the usergroups the current logged-in user belongs to, and the defined roles:

```
{set-block scope=root variable=cache_ttl}0{/set-block}
<h2>{$node.name}</h2>
{let current_user=fetch(user,current_user)
  parent_nodes=$current_user.contentobject.parent_nodes
  my_roles=$current_user.roles }
{cache-block keys=array($current_user.contentobject_id,"usergroups")}

<ul>
{section name=mygroup loop=$parent_nodes}
<li>
  {let thisnode=fetch(content,node,hash(node_id,$mygroup:item))}
  {$mygroup:thisnode.name} ({mygroup:thisnode.url_alias})
  {/let}
</li>
{/section}
</ul>
<h2>Assigned roles</h2>
<ul>
```

```

{section name=myrole loop=$my_roles}
<li>{$myrole:item.name} </li>
{/section}
</ul>
{/let}

```

Miscellaneous

Finally, here are some miscellaneous useful template tasks.

Show a Version History Audit Trail

To show an audit trail in view templates, you will have to control the cache lifetime for your view templates. This is best done by turning off view caching in the template file itself with the following code fragment at the top of your view template file:

```
{set-block scope=root variable=cache_ttl}0{/set-block}
```

Alternatively, you can also disable view caching, but this is not recommended because this will affect all your view templates. To compensate for the performance loss, cache blocks are used with relevant keys.

The following code extract looks for a user preference variable `audittrail`. You may omit this code, but then the audit trail will always be shown.

```

{let showaudit=ezpreference('audittrail')}
{cache-block
keys=array($node.object.modified,$node.node_id,$showaudit)}
{section show=eq(ezpreference('audittrail'),'on')}
<div style="font-size: small; background: lightyellow;
border: 1px solid blue;">
<h3>Audit trail (document history)</h3>
<table width="100%">
<tr><th><p>Version</p></th>
<th><p>Modified by</p></th>
<th><p>Date</p></th>
</tr>
{section name=versions loop=$node.object.versions
sequence=array(lightgrey,lightblue)}
<tr style="background: {$versions:sequence};">
<td><p>{$:item.version}</p></td>
<td><p>{$:item.creator.name}</p></td>
<td><p>{$:item.modified|l10n(shortdatetime)}</p></td>
</tr>
{/section}
</table>
</div>
<p style="text-align: right;"><a
href={"/user/preferences/set/audittrail/off"|ezurl}>Hide audit
trail</a></p>
{section-else}
<p style="text-align: right;"><a
href={"/user/preferences/set/audittrail/on"|ezurl}>Show audit
trail</a></p>

```

```
{/section}  
{/cache-block}  
{/let}
```

Show Creator, Modifier, and Publishing Date

The following code snippet displays the original author, the publishing date as well as the last modified date, and the user who modified the content:

```
<p class="date" style="text-align : right;">(Published:  
{ $node.object.published|l10n(shortdate)}, last modified:  
{ $node.object.modified|l10n(shortdatetime)})  
</p>  
<p class="date" style="text-align : right;">(Owner:  
{ $node.object.owner.name}, last edited by:  
{ $node.contentobject_version_object.creator.name})  
</p>
```

Listing keywords and their Automatically Related Objects

The following code example can be used in view mode. The view caching is first disabled at the top of the view template with:

```
{set-block scope=root variable=cache_ttl}0{/set-block}
```

After this, surround your normal attribute list with a cache block (with the node ID for the keys parameter) and add the following code to display the list of keyword-related objects of the same class:

```
<h2 class="title">Keywords</h2>  
{cache-block  
keys=array($node.object.data_map.keywords.content.keyword_string,  
           $node.node_id)}  
{let related=$node.object.data_map.keywords.content.related_objects}  
  {section show=$related}  
    <h3 class="luna">Related documents containing one or more of the  
same keywords</h3>  
    <ol>  
      {section name=Related loop=$related}  
        <li><a href={$:item.url_alias|ezurl}>{$:item.name}</a></li>  
      {/section}  
    </ol>  
  {/section}  
{/let}  
<br />  
{/cache-block}
```

Advanced Keyword Facility

The normal keyword list provided by the keyword attribute structure may be a bit too limiting. For example, the related objects are all of the same class as the node they are derived from.

You can create a dummy node somewhere in your node tree that does more than displaying its content. The strategy is simple, and consists of overriding the template for

the keyword datatype and providing links to the node that will service the advanced keyword functionality.

The override template for the keyword attribute is placed as a cascading template inside the design-specific templates:

```
design/<yourdesign>/templates/content/datatype/view/keyword.tpl
```

The content of keyword.tpl becomes:

```
{let kwarr=$attribute.content.keyword_string|explode(',')}
{section name=kw loop=$kwarr}
  <a href={concat('/home/services/keyword/(keyword)/',
    $kw:item|trim(' ')|urlencode)|ezurl}>{$kw:item}</a>
  {delimiter} | {/delimiter}
{/section}
{/let}
```

In this template code, /home/services/keyword is the special node for which we will create an override template. The next part of the URL is the custom template variable (available in eZ publish 3.4) with the name keyword and with a value of the actual keyword properly trimmed and URL encoded.

In an override template, you can put the following code, which will be the template code for the /home/services/keyword/ node:

```
{set-block scope=root variable=cache_ttl}0{/set-block}
<h1>Keywords</h1>
<p>{$view_parameters.keyword|urlencode}</p>
{* if the keyword is in fact a phrase, split it and allow for more
drill-down of the keywords *}
{section show=$view_parameters.keyword|count_words|gt(1)}
<p>Maybe try the compound words too:
  {section loop=$view_parameters.keyword|urlencode|explode(' ')}
    <a
href={concat('/home/services/keyword/(keyword)/', $:item)|ezurl}>
{$:item}</a>{delimiter} | {/delimiter}
  {/section}</p>
{/section}
{* do we really have a keyword from the url? *}
{section show=count($view_parameters.keyword)}
{* url decode needed for example to put back special characters *}
<h2>Keywords containing {$view_parameters.keyword|urlencode}</h2>
  {let keynodes=fetch(content, keyword, hash(alphabet, concat('%',
    $view_parameters.keyword|urlencode)))}
  {section name=kwlist loop=$keynodes}
  {* the keyword list with objects/nodes will only for the first item
in the keyword group contain the actual keyword. We will use this
fact to build our list. Counting the characters does the trick*}
  {section show=count($kwlist:item.keyword)}<h2>Matching objects for
keyword <span
style="color:red;">{$kwlist:item.keyword|urlencode}</span></h2>
  {/section}
  <p><a href={$kwlist:item.link_object.url_alias|ezurl}
title="{ $kwlist:item.link_object.name}">
{$kwlist:item.link_object.name|shorten(70)}</a>
```

```
</p>
{/section}
{/let}
{section-else}
<p>No keyword specified</p>
{/section}
```

Creating a Threaded Forum Template

This is simple: use the depth of the current child node relative to the top-level node in the forum to indent the item.

Suppose `$node` contains the main forum message object and `$Child` the array of child nodes of this main forum message. You can then use the following inside the loop to create an indented, threaded display:

```
{* inside loop for child forum messages *}
{let usedepth=sub($Child:item.object.main_node.depth,
                 $node.object.main_node.depth,1)}
<div style="margin-left: {$usedepth}em;">
  {node_view_gui content_node=$Child:item view="line"}
</div>
```

Summary

In this chapter, we have looked at eZ publish's powerful template engine, which is used to extract and present content to the user. To make use of the template engine, the language of the template system has to be learned.

We looked at the general principles of the template system, at page layout templates and content views, attribute templates, and edit templates. We saw how eZ publish uses caching to store template output for improved performance, and the template override system that allows a particular template to be used for a specific node, the depth in the site tree, the class of the content object, or other parameters.

We moved on to look at the basics of the template language—working with variables, and the constructs for controlling the flow of template output. We had a detailed look at the template functions available, and how to fetch, filter, and sort node lists, and display user information. The final part of the chapter showed how to implement common template tasks.

The eZ publish template engine is a very powerful beast, and in this chapter, we have only scratched the surface. You will find a reference to many of the template operators in *Appendix A: Operators and Functions*.

4

A Glimpse Inside the Core

eZ publish is not just another content management system that only allows you to add and modify content from an administrative interface—it actually provides a framework for creating your own extensions to the system, and allows you to work programmatically with many of the objects that you have seen over the last few chapters.

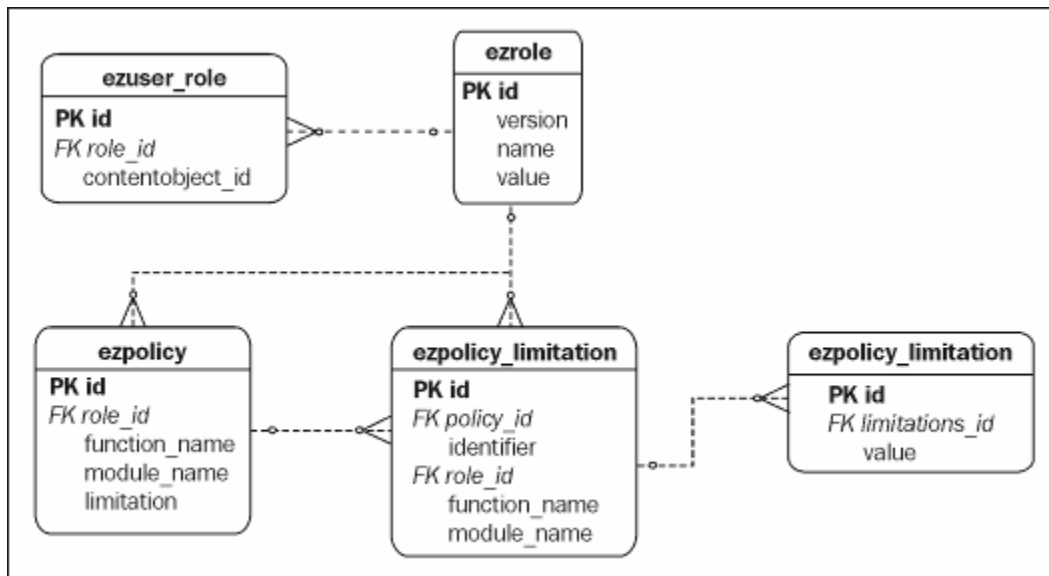
There is an extensive **object model** at the heart of eZ publish's core framework, and in this chapter, we take a look at some of the classes that relate to the concepts already discussed, and prepare you for creating your own extensions in the following chapters.

We also look at some of eZ publish's fundamental objects, the **kernel** classes, and see how they are realized at the code level. These classes, found in the `kernel/classes` folder, drive eZ publish's low-level functionality, and deal with activities such as persisting objects to the database, content handling, permissions and workflows, and datatypes.

Permissions

Restrictions on what a user can do are handled by roles in eZ publish. A **role** consists of one or more policies. A **policy** is related to a certain module and one or all functions of a module. Depending on the module function being applied to the policy, there will be certain policy limitation options that restrict or allow actions upon a section, node, or subtree, or other options that might be provided by the module definition.

The following database diagram shows the relations between objects, roles, permissions, and limitations:



Currently, the following limitations of policies are known to eZ publish. They are available for all modules. This list shows the actual identifiers for the limitation:

- Class
- Section
- Owner
- Status
- Node
- Subtree
- ParentClass

Here is how you set permissions in the module definition—the next chapter contains more information on module definitions.

```
$FunctionList['read'] = array( 'Class' => $ClassID,
                              'Section' => $SectionID,
                              'Owner' => $Assigned,
                              'Node' => $Node,
                              'Subtree' => $Subtree);
```

The following example gives an overview of how the main execution script can determine whether a user is allowed to access the module functionality.

The first thing that needs to be done is identify the user. When you have a user ID, you will know which roles this user has. From these roles, you can obtain the permission list:

```
include_once( "kernel/classes/datatypes/ezuser/ezuser.php" );
$currentUser =& eZUser::currentUser();
$accessResult = $currentUser->hasAccessTo($module->attribute('name'),
                                          $runningFunctions[0] );
```

The `hasAccessTo()` function returns information about the role, based on the name of the module and the function the user is attempting to access. `hasAccessTo()` returns an array with two keys of data, the `accessResult`. The `accessWord` will have the values `yes`, `no`, or `limited`. If the `accessWord` is `limited`, another key, `policies`, will be filled with a list of objects of `eZPolicy` that apply to that user. eZ publish will take one of the following actions based on the result:

- If the access result is `yes`, grant full execution rights to the user.
- If the result is `limited`, assign the list of policies to a global variable. Policies are individually handled for every module function.
- If the result is `no`, deny access to the user.

We will now look at a limitation check inside an eZ publish module. Actually, these limitations are not controlled by the module—they are controlled by the classes that handle content objects, such as `eZContentObject`, `eZContentObjectTreeNode`, and `eZSearchEngine`.

You should be able to find the limitations in the following global variable. However, they are submitted as a parameter to the module. Consider:

```
$GLOBALS['ezpolicylimitation_list']
```

And

```
$params['Limitation']
```

The module parameter has a higher priority than the global variable.

This following code demonstrates how `eZContentObjectTreeNode::subTreeCount()` handles the limitation by building the proper SQL query. Only the content objects that match the limitation are fetched.

```
if ( count( $limitationList ) > 0 )
{
    $sqlParts = array();
    foreach( $limitationList as $limitationArray )
    {
        $sqlPartPart = array();
        $hasNodeLimitation = false;
        foreach ( $limitationArray as $limitation )
        {
            if ( $limitation->attribute( 'identifier' ) == 'Class' )
            {
                $sqlPartPart[] = 'ezcontentobject.contentclass_id in (\'
                    $limitation->attribute( 'values_as_string' ).'\)';
            }
        }
    }
}
```

```
elseif ( $limitation->attribute( 'identifier' ) == 'Section' )
{
    $sqlPartPart[] = 'ezcontentobject.section_id in ('.
        $limitation->attribute( 'values_as_string' ) . ')';
}
elseif( $limitation->attribute( 'identifier' ) == 'Owner' )
{
    user =& eZUser::currentUser();
    $userID = $user->attribute( 'contentobject_id' );
    $sqlPartPart[] = "ezcontentobject.owner_id = '" .
        $db->escapeString( $userID ) . "'";
}
```

The following error codes are used by the system. They mostly apply to access and permission issues.

- Access denied (1)
- Object not found (2)
- Object not available (3)
- Module not found (20)
- Module view not found (21)
- Module or view disabled (22)
- No DB connection (50)

Object Persistence

eZ publish features a **persistent object** model for classes to store their data in the database. Providing metadata that links the code object to the corresponding database table makes it straightforward to store, fetch, or delete an object without having to explicitly construct SQL statements. This functionality is achieved through the `eZPersistentObject` class.

The `eZPersistentObject` class can be viewed as a database abstraction layer between the application and the eZDB database interface. By creating a class that inherits from `eZPersistentObject`, just like the `eZContentClass` and `eZContentObject` classes, you obtain all the data-access functionality for free, and after specifying the object metadata, you have an easily persistable object.

For such persistable objects, there is a uniform way to access their fields or methods and also to provide object metadata—**attributes**. Attributes provide an easy mapping between database fields and the equivalent fields of the object representing the data. They also provide the mapping to member functions of the object, which allow for setting of data that goes beyond simple values stored in object fields.

The attributes of an object are defined and returned by its `definition()` function. Each class that inherits from `eZPersistentObject` will need to implement this function to provide object metadata.

The `definition()` function returns (a reference to) an associative array that includes the following information:

Value	Description
<code>Fields</code>	An associative array of field mappings, mapping the database field to the object field. This array also contains metadata about the datatype of the field, the default value, and whether the field is required.
<code>function_attributes</code>	An associative array of attributes that map to member functions, used for fetching data with functions.
<code>set_functions</code>	An associative array of attributes that map to member functions, used for setting data with functions.
<code>keys</code>	An array of fields that are the keys of the database table, and are used to identify a row in the table.
<code>increment_key</code>	The name of the field incremented on table inserts.
<code>class_name</code>	The class name used for instantiating new objects when fetching from the database.
<code>name</code>	The name of the underlying database table.

By way of an example, here is a class that manages a list of names stored in the `eznames` database table with the two columns `id` and `name`:

```
class eZNames extends eZPersistentObject
{
    function eZNames( $row = array() )
    {
        $this->eZPersistentObject( $row );
    }
    function &definition()
    {
        return array( "fields" => array(
            "id" => array( 'name' => 'ID',
                        'datatype' => 'integer',
                        'default' => 0,
                        'required' => true ),
            "name" => array( 'name' => "Name",
                        'datatype' => 'string',
                        'default' => '',
                        'required' => true ) ),
            "keys" => array( "id" ),
```

```
        "increment_key" => "id",
        "sort" => array( "name" => "asc" ),
        "class_name" => "eZNames",
        "name" => "eznames" );
    }
}
```

The two database fields, `id` and `name`, map to the object fields `ID` and `Name` respectively, as indicated by the `name` value of their array.

Getting Attribute Values

The `attribute()` function is used to get the value of an attribute by passing in the name of the attribute. This function returns the value of the member function or field corresponding to the attribute. The `attribute()` function is provided by the `eZPersistentObject` base class.

Note that some classes provide their own implementation that explicitly checks the name of the attribute and calls a specific function based on the name, and passes to the base class implementation for other attributes. For example, here is the `attribute()` implementation in the `eZBasket` class:

```
function attribute( $attr )
{
    if ( $attr == "items" )
        return $this->items();
    else if ( $attr == "total_ex_vat" )
        return $this->totalExVAT();
    else
        return eZPersistentObject::attribute( $attr );
}
```

This class does not have any mapping between database fields and member functions defined in the `definition()` function, and chooses to handle it here in the `attribute()` method.

Setting Attribute Values

Attribute values are set through the `setAttribute()` function of the `eZPersistentObject` base class, by passing in the name of the attribute and the value to store. Once again, this function determines which member function or field is to be used to hold the value.

Other Attribute Functions

Two other functions in `eZPersistentObject` complete the attribute handling functionality: `attributes()` and `hasAttribute()`.

The `attributes()` function returns an array that contains the name of all the attributes, both field and function mappings. One thing to note about the `eZBasket` situation is that the attributes listed in its `attribute()` function would not be returned in the array from `attributes()`—this function returns the attributes specified in the `definition()` function.

The `hasAttribute()` function indicates whether the specified attribute is defined for that object.

Another thing to note about the `definition()` function is that if you inherit from a class that already provides a `definition()` function, your new implementation of `definition()` will override the list of attributes from the parent class.

Persistent Storage

Through the use of attributes and the metadata specified by the `definition()` function, the `eZPersistentObject` class can:

- Fetch a fully-populated object (or list of objects) representing data stored in the relevant database table
- Store a populated object into the appropriate fields in the relevant database table

Fetching Data

Single objects can be fetched with the `fetchObject()` function. This calls the `fetchObjectList()` function, which fetches a collection of objects, but `fetchObject()` only takes the first result from this list. For example, to fetch the `name` field for a given record from the `eznames` table, we would use:

```
$object->fetchObject(eZNames::definition(),array('id' =>$id))
$name = $object->attribute('name');
```

Storing Data

Using all the metadata provided by the `definition()` function, `eZPersistentObject` can generate the SQL statement required to persist the object to the database. It can also work out if this data needs to be an `INSERT` or an `UPDATE`, so a call to `store()` removes much of the data-access code that is usually required for saving objects to the database.

The following code inserts a new value in a table and gets the ID:

```
$object = & new eZNames(array('name'='Bjoern'));
$object->store();
$id = $object->attribute('id');
```

Other Data Manipulation

eZPersistentObject has other methods for general manipulation of data, such as deleting or moving rows within the database.

For example, to remove a row from our eznames table by specifying the ID of the row, you can use:

```
$object->remove(array('id' => $id));
```

Now that you have seen how eZ publish persists object to the database, let's move on to look at how content classes and content objects are handled by the system.

Content Classes

A content class in eZ publish is represented by the eZContentClass class. This class offers methods for creating, modifying, deleting, and fetching content classes. Content classes usually have multiple content class attributes assigned, and these are in turn managed by the eZContentClassAttribute class.

In the code download for this chapter, you will find a script called `create.php` that creates a content class with two content class attributes: the datatypes `ezstring` and `ezimage`. In this section, we will walk through the important points of the process using this example as an illustration.

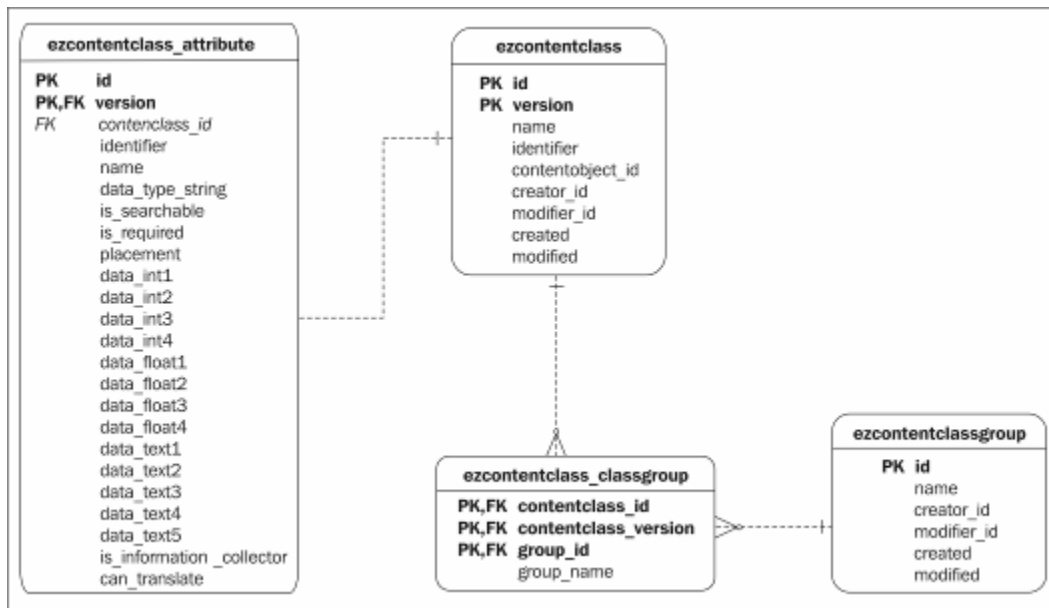
The steps to create a content class are:

1. Create a new empty eZContentClass instance.
2. Define and add content class attributes.

Each content class has two important properties that are used throughout the system. The first is the **identifier**. This is a name pattern that can be used instead of the numerical ID. Always try to use your named identifiers before you use content object class IDs. This will make your code more portable and easier to maintain. The other important property is the object name. The name of an object can consist of a list of strings and values of content object attributes.

A class can have three states that can be defined when creating new classes:

- Current active class: `EZ_CLASS_VERSION_STATUS_DEFINED`
- Draft of a class: `EZ_CLASS_VERSION_STATUS_TEMPORARY`
- Modified class not active: `EZ_CLASS_VERSION_STATUS_MODIFIED`



To begin creating a new content class, we get a new instance of `ezContentClass`, fetching the current user to be the object owner:

```
$user =& ezUser::fetch($user_id);
$user_id = $user->attribute( 'contentobject_id' );
$class =& ezContentClass::create( $user_id );
```

Next, we set the class's attributes, its version (the version is always 0 for a content class), name, identifier, and the pattern for content objects that can be created from this class:

```
$class->setAttribute( 'version', 0 );
$class->setAttribute( 'name', 'My Custom Class' );
$class->setAttribute( 'identifier', 'custom_class' );
$class->setAttribute( 'contentobject_name', 'Custom Class <id>' );
```

Next, we store the class:

```
$class->store();
```

Finally, we assign the content class to a content class group. We will assign it to the group `Content`, which has a group ID of 1. We also need the class ID and the version of the class:

```
$classID = $class->attribute( 'id' );
$classVersion = $class->attribute( 'version' );
$ingroup =& ezContentClassClassGroup::create( $classID,
    $classVersion, GroupID, $GroupName );
$ingroup->store();
```

Now that we have created our basic content class, we can add content class attributes.

Content Class Attributes

Content class attributes are represented by the `eZContentClassAttribute` class. To create a content class attribute for a content class, we specify the class ID and the datatype of the content class attribute.

In the previous chapters, we saw the datatypes that ship with eZ publish. Datatypes are actually classes that inherit from `eZDataType`, and the standard ones can be found in the `kernel/classes/datatypes` folder.

Continuing our example, we will create two content class attributes for our content class: one of type `ezstring` and the other of type `ezimage`. First, the `ezstring` type:

```
$DataTypeString = 'ezstring';
$new_attribute =& eZContentClassAttribute::create( $ClassID,
                                                $DataTypeString );
```

We need to set the version, name, and identifier for this content class attribute:

```
$new_attribute->setAttribute( 'version', $ClassVersion);
$new_attribute->setAttribute( 'name', 'new attribute
                              '.$DataTypeString );
$new_attribute->setAttribute( 'identifier',
                              'new_identifier_'.$DataTypeString );
```

The datatype can initialize the content class attribute (this is not the same as providing a default value for the content object attribute):

```
$dataType = $new_attribute->dataType();
$dataType->initializeClassAttribute( $new_attribute );
```

The new attribute is stored to the database:

```
$new_attribute->store();
```

We follow a similar process to add a content class attribute of type `ezimage`:

```
$DataTypeString = 'ezimage'
$new_attribute =& eZContentClassAttribute::create( $ClassID,
                                                $DataTypeString );

$new_attribute->setAttribute( 'version', $ClassVersion);
$new_attribute->setAttribute( 'name', 'new attribute
                              '.$DataTypeString );
$new_attribute->setAttribute( 'identifier',
                              'new_identifier_'.$DataTypeString );

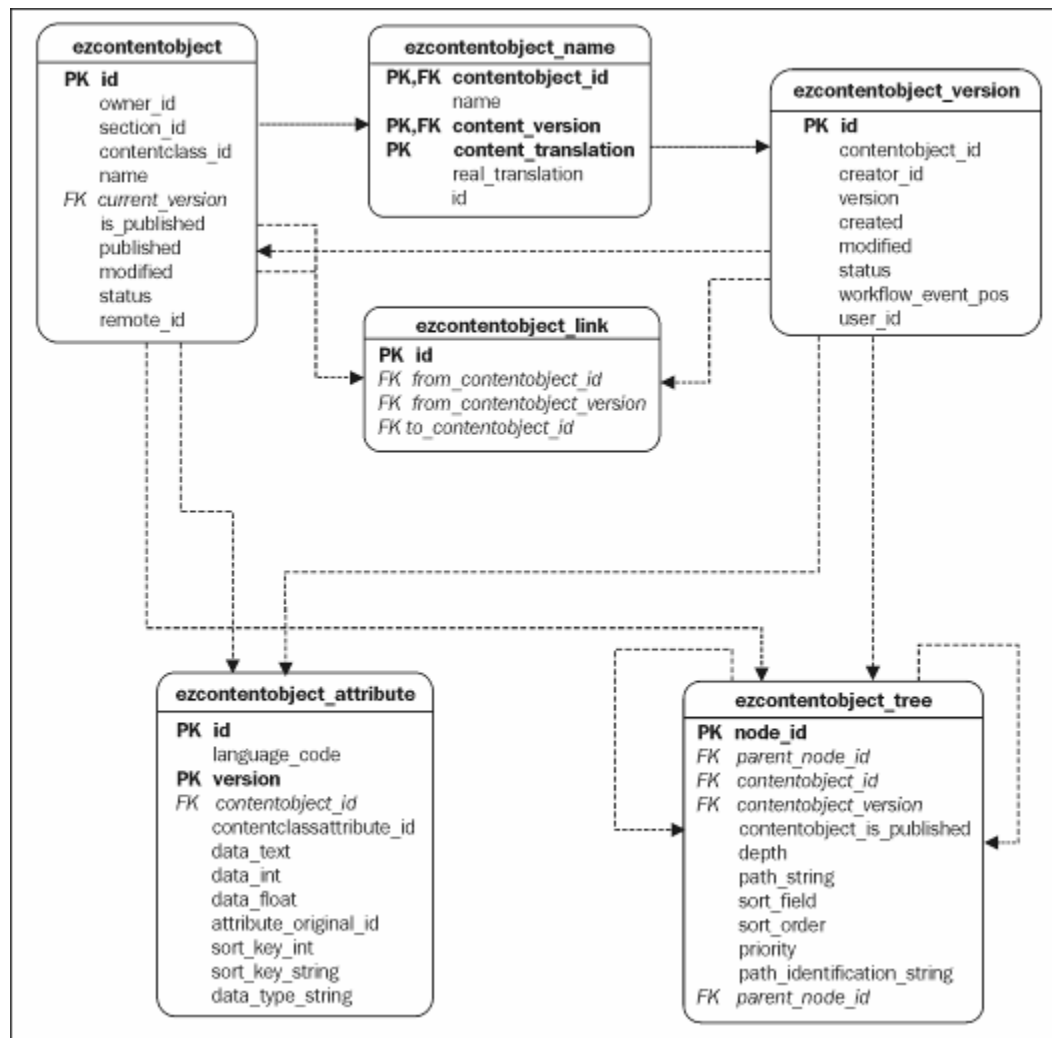
$dataType = $new_attribute->dataType();
$dataType->initializeClassAttribute( $new_attribute );
$new_attribute->store();
```

Now that the content class attributes are in place, we can proceed with creating content objects from the content class.

Content Objects

In this part of the book, we look at handling content objects of eZ publish at a low level. With the help of these objects, you can define your site tree. A content object is assigned to a certain user and section. Usually, every object is assigned to one or more nodes out of the content object tree.

Content objects are represented by the eZContentObject class, and the following diagram shows the database tables that relate to the underlying ezcontentobject content object table:



Creating a Content Object

An object definition is needed before an actual content object can be populated. Content objects are defined through content classes. Further on, we will also need to assign a certain node or placement to this object. This placement is called the parent node. The parent node has a unique node ID like any other node in the system. A content object has three states it can be assigned to:

- EZ_CONTENT_OBJECT_STATUS_DRAFT
- EZ_CONTENT_OBJECT_STATUS_PUBLISHED
- EZ_CONTENT_OBJECT_STATUS_ARCHIVED

To create and populate a content object, we need to go through the following steps:

1. Create a content object instance from its content class.
2. Create content object node assignments.
3. Store the empty content object.
4. Fill the content object attributes.
5. Store the filled content object.
6. Publish the object if needed.

We will now go into the process of creating a new content object from a certain class. In the first step, we will need to instantiate an object from the content object class.

Create a Content Object Instance

The requirements to create a new content object are a system user and a section ID. The system user will be the owner of the content object. For creating a node assignment, we will also need a related parent object. We can draw the objects section from the parent object.

```
$parentNodeID=2;
$class = eZContentClass::fetchByIdentifier('custom_class');
$parentContentObjectTreeNode =
    eZContentObjectTreeNode::fetch($parentNodeID);
$parentContentObject = $parentContentObjectTreeNode-
    >attribute("object");
$sectionID = $parentContentObject->attribute( 'section_id' );
$contentObject =& $class->instantiate( $user_id, $sectionID );
```

Content Object Node Assignments

In this step, we will place the empty object inside the site tree. This node will be also the main node of the content object. To properly assign this content object, we also need to set a parent node to this object.

```
$nodeAssignment =& eZNodeAssignment::create( array(
    'contentobject_id' => $contentObject->attribute('id'),
    'contentobject_version' => $contentObject-
        >attribute('current_version' ),
```

```
'parent_node' => $parentContentObjectTreeNode->attribute
('node_id' ),
'is_main' => 1));
```

Now we can store the information about this node in the content object tree:

```
$nodeAssignment->store();
```

You should always create the first node assignment before you enter any data into the object.

Before we add the content class attributes, we need to define the name attribute of the content object:

```
$contentObject->setAttribute( 'name', 'Custom Class Number One' );
```

We will now store the current object in the database, before adding any of the content class attributes:

```
$contentObject->store();
```

Content Object Attributes

We will now save data for every content object attribute the content class provides to our content object.

First, we need to get all content object attributes.

```
$attrs =& $contentObject->contentObjectAttributes();
```

We will loop through each attribute and process each one depending on the identifier.

```
$loopLength = count($attrs);
for($i=0;$i<$loopLength;$i++)
{
    switch($attrs[$i]->attribute(
        "contentclass_attribute_identifier"))
    {
        case 'new_identifier_ezstring':
            $attrs[$i]->setAttribute('data_text', "Some input string");
            $attrs[$i]->store();

            break;

        case 'new_identifier_ezimage':
            $content =& $attrs[$i]->attribute( 'content' );

            //Image init data

            $imageAltText="Alternate Text";

            //name on disk

            $filename="logo.gif";
```

```
        //how we name it
        $originalFilename="logo.gif";
        $content->initializeFromFile($filename,
                                     $imageAltText,
                                     $originalFilename);
        $content->store();
        $attrs[$i]->store();
        break;
    }
}
```

The main attributes of `ezContentObjectAttribute` that hold data are:

- `data_float`
- `data_int`
- `data_text`

In the final step, we set the status of the content object, and make it available as a draft for the user we selected before.

```
$contentObject->setAttribute( 'status', EZ_VERSION_STATUS_DRAFT );
$contentObject->store();
```

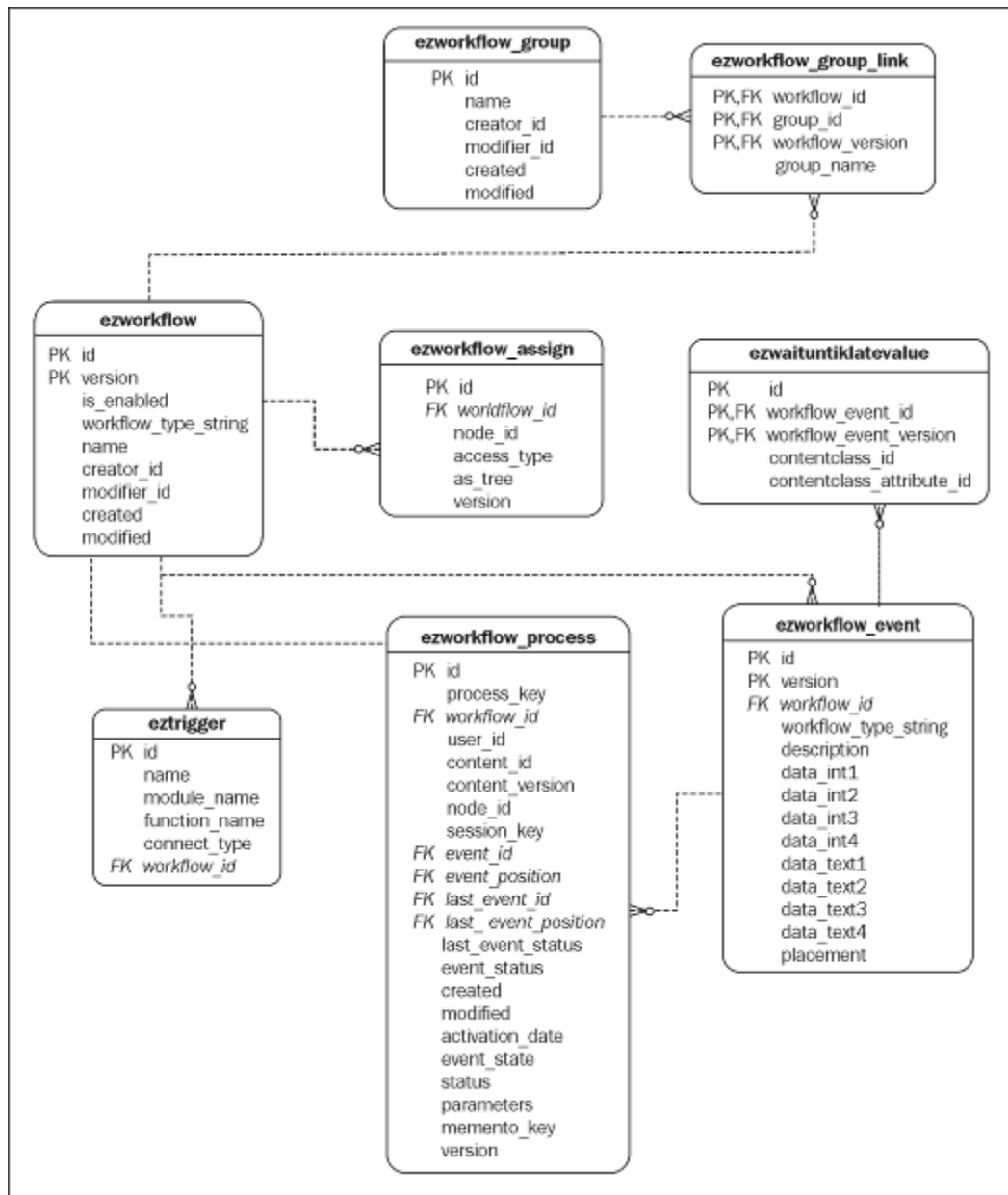
It could be also nice to run some operation on this object. We could, for example, publish it:

```
$operationResult = ezOperationHandler::execute('content', 'publish',
        array( 'object_id' =>
                $contentObject->attribute( 'id' ),
                'version' => 1 ) );
```

Executing this operation will execute a group of other functions and triggers.

Workflows and Triggers

The following diagram shows the main database tables and the relations between them that handle the interplay between objects, workflows, and triggers:



Workflows are triggered by the operation definition of the operation. The triggers are defined in the `operation_definition.php` file of each module:

```

$operationList['publish'] = array( 'name' => 'publish', 'body' =>
array(
    'type' => 'trigger',

```

```
        'name' => 'pre_publish',
        'keys' => array( 'object_id', 'version' )
    )
)

array( 'type' => 'method',
      'name' => 'set-object-published',
      'frequency' => 'once',
      'method' => 'setObjectStatusPublished',
      'parameters' => array( array( 'name' => 'object_id',
      'type' => 'integer',
      'required' => true ),
    )
}
```

An operation consists of triggers and methods. For example, the operation could be processed in this way:

- Run the first trigger before anything else (trigger)
- Do something useful (method)
- Run the second trigger (trigger)
- Do something else (method)
- Run the last trigger at the end of the operation (trigger)

We find an example in the content/edit/ module, which corresponds to the file content/edit.php. When this example is executed, a content object will be published. We will now look closer into the process of running a workflow connected to the pre_publish trigger.

```
if ( $module->isCurrentAction( 'Publish' ) ){
    include_once( 'lib/ezutils/classes/ezoperationhandler.php' );
    $operationResult = ezOperationHandler::execute(
        'content', //module
        'publish', //operation
        array(
            'object_id' => $object->attribute( 'id' ),
            'version' => $version->attribute( 'version' ) ) );
}
```

ezOperationHandler() is now executing the publish operation out of the content module on a certain version of a content object. ezOperationHandler() has a helper, the ezModuleOperationInfo() function, which switches control of the process of execution and fails if the operation definition is not properly set. The utilities that deal with operation handling are found in the ezUtils library.

Each trigger will return the status of the current operation. eZ publish currently knows three statuses:

- EZ_MODULE_OPERATION_CANCELED
- EZ_MODULE_OPERATION_HALTED
- EZ_MODULE_OPERATION_CONTINUE

The status tells eZ publish whether the operation is canceled, stopped temporarily, or can be continued. If the operation is executed again on the same object and version, it will continue from where it had stopped the last time. The following code shows the `pre_publish` trigger being executed:

```
$status = eZTrigger::runTrigger(
    $triggerName, $this->ModuleName, $operationName,
    $operationParameters, $triggerKeys);
```

eZ publish provides the following return statuses for triggers:

- `EZ_TRIGGER_STATUS_CRON_JOB`
- `EZ_TRIGGER_WORKFLOW_DONE`
- `EZ_TRIGGER_WORKFLOW_CANCELED`
- `EZ_TRIGGER_NO_CONNECTED_WORKFLOWS`
- `EZ_TRIGGER_FETCH_TEMPLATE`
- `EZ_TRIGGER_REDIRECT`
- `EZ_TRIGGER_WORKFLOW_RESET`

A workflow is processed when it is connected to a trigger. If the workflow hasn't started earlier, eZ publish will now create the new workflow process; otherwise it will process the existing one. The identifier for a workflow process is the process key.

```
$processKey = eZWorkflowProcess::createKey( $parameters, $keys );
$workflowProcessList =& eZWorkflowProcess::fetchListByKey(
    $processKey );
```

If the fetch returns no result, we can create a new one:

```
$workflowProcess =& eZWorkflowProcess::create( $processKey,
    $parameters );
$workflowProcess->store();
return eZTrigger::runWorkflow( $workflowProcess );
```

eZ publish provides the following return statuses for Workflows:

- `EZ_WORKFLOW_STATUS_NONE`
- `EZ_WORKFLOW_STATUS_BUSY`
- `EZ_WORKFLOW_STATUS_DONE`
- `EZ_WORKFLOW_STATUS_FAILED`
- `EZ_WORKFLOW_STATUS_DEFERRED_TO_CRON`
- `EZ_WORKFLOW_STATUS_CANCELLED`
- `EZ_WORKFLOW_STATUS_FETCH_TEMPLATE`
- `EZ_WORKFLOW_STATUS_REDIRECT`
- `EZ_WORKFLOW_STATUS_RESET`

When running the workflow, eZ publish will process all workflow events that are assigned to the workflow in a loop until a proper return has been found. Each time the loop starts over, an event of the workflow is processed.

An `eZWorkflowType` should return one of the following—watch your returns on module development:

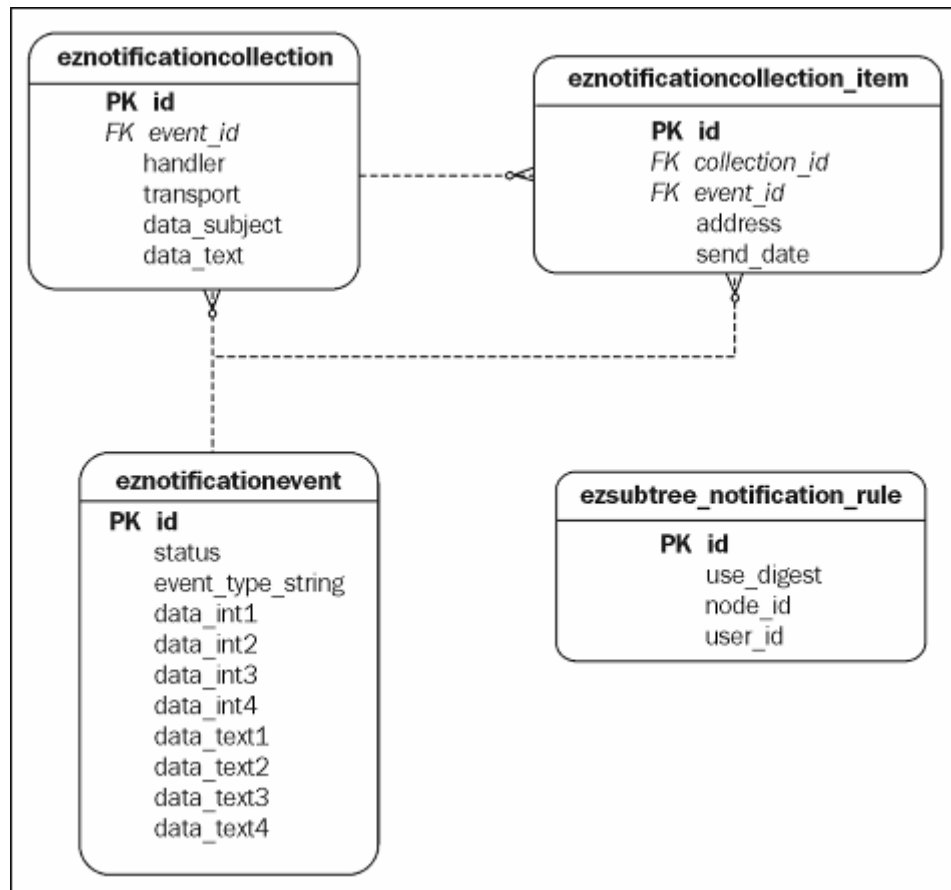
- `EZ_WORKFLOW_TYPE_STATUS_NONE`
- `EZ_WORKFLOW_TYPE_STATUS_ACCEPTED`
- `EZ_WORKFLOW_TYPE_STATUS_REJECTED`
- `EZ_WORKFLOW_TYPE_STATUS_DEFERRED_TO_CRON`
- `EZ_WORKFLOW_TYPE_STATUS_DEFERRED_TO_CRON_REPEAT`
- `EZ_WORKFLOW_TYPE_STATUS_RUN_SUB_EVENT`
- `EZ_WORKFLOW_TYPE_STATUS_WORKFLOW_CANCELLED`
- `EZ_WORKFLOW_TYPE_STATUS_FETCH_TEMPLATE`
- `EZ_WORKFLOW_TYPE_STATUS_FETCH_TEMPLATE_REPEAT`
- `EZ_WORKFLOW_TYPE_STATUS_REDIRECT`
- `EZ_WORKFLOW_TYPE_STATUS_WORKFLOW_DONE`
- `EZ_WORKFLOW_TYPE_STATUS_REDIRECT_REPEAT`
- `EZ_WORKFLOW_TYPE_STATUS_WORKFLOW_RESET`

Notifications

eZ publish has a built-in notification system. On certain events, a system user is able to receive a notification about this event. The events are based on a plug-in implementation. The following events are known to the eZ publish system by default:

- Collaboration
- Publish
- Current time

The following sub-diagram of the database shows the relations between content objects and the notification system:



Notification events are placed inside the `kernel/classes/notification` folder.

There are several handlers that process notification events. The following handlers are known to the system by default:

- Collaboration notification
- General digest
- Subtree notification

We will look at the process of the subtree notification on a publish event.

Each time a content object or content object version is published through the content module, the `publish` operation is used:

```

$operationResult = ezOperationHandler::execute(
    'content',
    'publish',
    array

```

```
( 'object_id' => $contentObject->attribute( 'id' ),  
  'version' => $contentObject->attribute( 'current_version' )  
);
```

The publish operation invokes a method call to `eZContentOperationCollection::createNotificationEvent()`:

```
array(  
  'type' => 'method',  
  'name' => 'create-notification',  
  'frequency' => 'once',  
  'method' => 'createNotificationEvent',  
),
```

This call will result in a new event of the type `ezpublish` being created and stored to the database.

```
include_once( 'kernel/classes/notification/eznotificationevent.php'  
);  
$event = & eZNotificationEvent::create  
(  
  'ezpublish',  
  array  
  (  
    'object' => $objectID,  
    'version' => $versionNum  
  )  
);  
$event->store();
```

This event can be processed by zero, one, or more notification handlers.

`eZNotificationEventFilter` will start this process. If the handler knows how to handle a certain event that is identified by the `event_type_string`, it will be processed.

```
include_once( 'kernel/classes/notification/  
              eznotificationeventfilter.php' );  
eZNotificationEventFilter::process();
```

`eZ publish` will now loop through every unhandled event and try to run it with every available handler in the system. If every notification has been sent successfully, the event can now be deleted from the system.

```
$eventList =& eZNotificationEvent::fetchUnhandledList();  
$availableHandlers =&  
    eZNotificationEventFilter::availableHandlers();  
foreach( array_keys( $eventList ) as $key )  
{  
  $event =& $eventList[$key];  
  foreach( array_keys( $availableHandlers ) as $handlerKey )  
  {  
    $handler =& $availableHandlers[$handlerKey];  
    $handler->handle( $event );  
  }  
  $itemCountLeft =&  
    eZNotificationCollectionItem::fetchCountForEvent(  
        $event->attribute( 'id' ) );  
  if ( $itemCountLeft == 0 )
```

```

    {
        $sevent->remove();
    }
    else
    {
        $sevent->setAttribute( 'status',
                               EZ_NOTIFICATIONEVENT_STATUS_HANDLED );
        $sevent->store();
    }
}

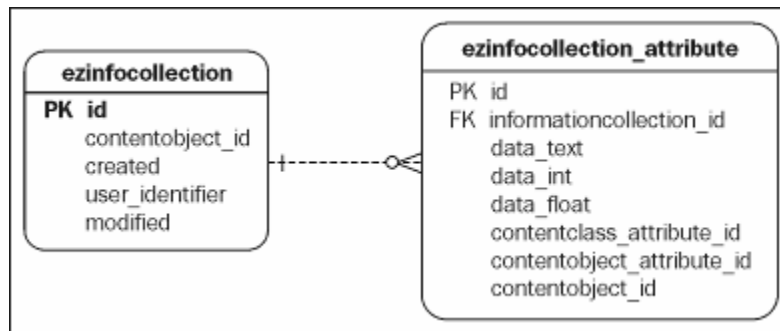
```

A handler can return the following statuses that describe the current state:

- EZ_NOTIFICATIONEVENTHANDLER_EVENT_HANDLED
- EZ_NOTIFICATIONEVENTHANDLER_EVENT_SKIPPED
- EZ_NOTIFICATIONEVENTHANDLER_EVENT_UNKNOWN
- EZ_NOTIFICATIONEVENTHANDLER_EVENT_ERROR

Information Collection

The eZ publish information collector can gather information or feedback about a content object. This information will be stored in the database and can be also sent via e-mail. The storage and presentation of a collected item is defined through the information collector type. This diagram shows the relations between the relevant database tables.



eZ publish comes with several information collector types by default.

- Poll
- Form
- Feedback

The information collector is setup through the `collect.ini` file and has various options that can be modified. A type can be also assigned through an attribute of a content object. By default, the attribute identifier is `collection_type`:

```
TypeAttribute=collection_type
```

In the collection setting, you can define how the submission of user information should work or what policies should apply.

`CollectAnonymousData` can be set to `enabled`, allowing an anonymous user to submit information to a type:

```
CollectAnonymousData=enabled
```

You can override each setting per type individually:

```
CollectAnonymousDataList[poll]=disabled
```

You can also set a content object attribute that will pass the setting to the information collector:

```
CollectAnonymousDataAttribute=collection_anonymous
```

eZ publish offers three policies on how user input is handled. Once again, just like `CollectAnonymousData`, you can set the definition globally, per type, or per content object:

- `multiple`: Each user can submit multiple data
- `unique`: One set of data per user, if data already exists give a warning
- `overwrite`: One set of data per user but new entry overwrites old one

Users are tracked by their user ID, and if they are anonymous to the system, their remote IP will be used to identify them.

Setting the global policy:

```
CollectionUserData=multiple
```

Setting the policy per type:

```
CollectionUserDataList[poll]=overwrite
```

Setting the policy per content object

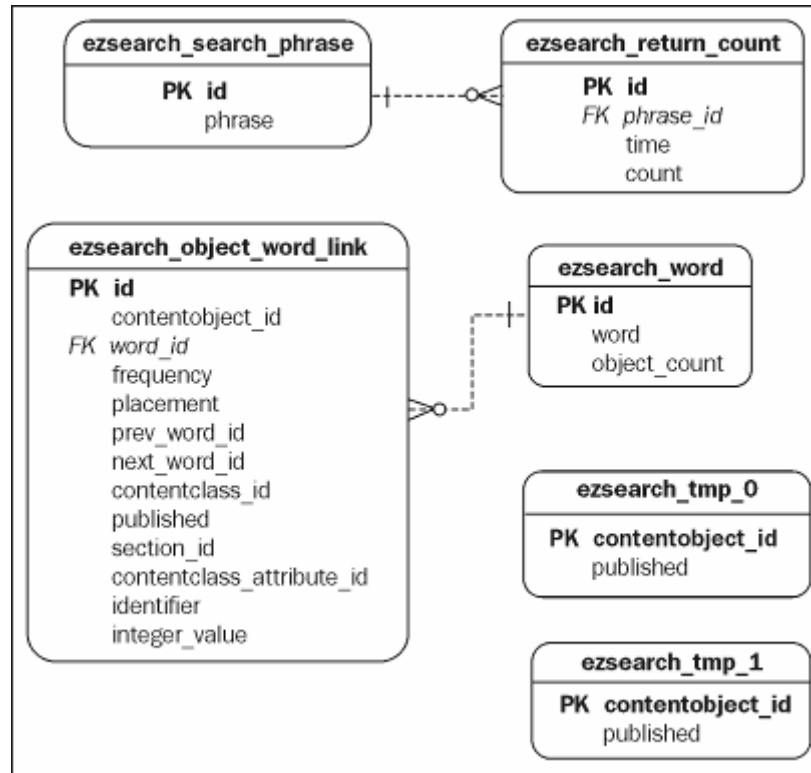
```
CollectionUserDataAttribute=collection_userdata
```

Searching

eZ publish offers an `eZSearch` class for including your own search engine. All the existing implementations are used as plug-ins to the system. They can be found under `kernel/search/plugins`. By default, eZ publish uses the `eZSearchEngine` plug-in. `OpenFTS` is also available as a plug-in, but it doesn't seem to be fully implemented. `OpenFTS` can be only used with PostgreSQL. Writing your own search engine is quite a lot of work.

The following database diagram shows the relations between objects and the `eZSearchEngine` search engine plug-in. The `ezsearch_tmp_0` and `ezsearch_tmp_1` tables are temporary tables, and are created on demand. Each temporary table gives the

result from one word of the search phrase. The result of a search in eZ publish is a combination of all temporary tables. The tables `ezsearch_search_phrase` and `ezsearch_return_count` are only important for the purpose of the search statistics (`kernel/search/stats.php`).



`eZSearch` (`kernel/classes/ezsearch.php`) is the interface to the plug-in. You will find its setup in `site.ini`:

```
[SearchSettings]
SearchEngine=eZSearchEngine
#SearchEngine=openFts
SearchViewHandling=default
LogSearchStats=enabled
MaximumSearchLimit=30
AllowEmptySearch=disabled
```

Every time a content object is edited, the object must be deleted and inserted into the search again. Here is how the `kernel/content/edit.php` file uses the `eZSearch` interface:

```
eZSearch::removeObject( $object );
eZSearch::addObject( $object );
```

If a content object is removed or put into trash, only a single function is called:

```
eZSearch::removeObject( $object );
```

eZ publish offers two module views that you can use for searches. Both are very similar and work in the same way. Both perform their search over the interface of eZSearch (eZSearch::search()). The content/search view uses the fulltext search type by default, and the content/advancedsearch view uses a wide spectrum of search types and filters offered by the plug-in.

With those two steps, the advanced search will get the requested information from the eZSearch engine. eZSearch::buildSearchArray() prepares a search definition depending on the plug-in. eZSearch::search() returns an associative array with information about the result (array of eZContentObjectTreeNode, SearchCount, and StopWordArray).

```
$searchArray =& eZSearch::buildSearchArray();
$searchResult =& eZSearch::search(
    $searchText,
    array( 'SearchSectionID' => $searchSectionID,
          'SearchContentClassID' => $searchContentClassID,
          'SearchContentClassAttributeID' =>
            $searchContentClassAttributeID,
          'SearchSubTreeArray' => $subTreeArray,
          'SearchDate' => $searchDate,
          'SearchTimestamp' => $searchTimestamp,
          'SearchLimit' => $pageLimit,
          'SearchOffset' => $offset ),
    $searchArray );
```

The available general search filters for eZSearchEngine are:

- class: Filters a content object class
- publishdate: Filters a certain creation date
- subtree: Filters a certain subtree
- section: Filters a certain section
- offset: Where to start with the result
- limit: Where to end the result

Summary

In this chapter, we took a brief look at realizing eZ publish fundamentals at the code level, and how objects are persisted to the database.

The next chapter looks at the process of creating an eZ publish extension.

5

Extending eZ publish

The eZ publish system contains a collection of modules and libraries that together provide a wide range of functionality for users. In order for eZ publish to address the needs of the growing CMS market, users must be able to add functionality of their own. This is achieved by using custom-designed **extensions**.

The modular nature of eZ is reinforced through the use of extensions. Extensions allow for template operators, datatypes, functions, and whole modules that can co-exist with one another to be developed separately and added to the eZ publish install. A functional extension may be something as small as a new template operator, or something as critical as an interface to your other business applications.

The aim of this chapter is to discuss how to use the extension mechanism. With almost every area of eZ publish open to the extension system, the areas covered by this chapter include:

- Modules
- Datatypes
- Template operators
- Workflow events
- Actions
- Notifications

We will also discuss examples of how to use eZ for data interoperability with other systems, including using SOAP and RSS as a means of facilitating the transfer of data.

Why Create an Extension?

Before embarking on creating an extension, you need to understand why you are creating an extension. It also is worth spending time researching eZ publish forums and reviewing the functionality that will be available in the next release of eZ publish. You may

discover that a feature is already available in the SVN code repository at <http://pubsvn.ez.no>.

There are several reasons why you may need to write an extension:

- eZ publish does not have the functionality you require.
- A similar but not exact feature may be available, in which case you can submit a feature request to the eZ team.
- The feature may have bugs, preventing it from being used as intended. This should be reported as a bug, but if time is short, a quick extension can be used as a temporary solution.
- The feature may be proprietary to your company and essential for use in your CMS.
- The feature may facilitate import/export functions for data from/to another system.

When naming your extensions, make sure you use a unique name, as this will prevent problems with duplicate names in the future. Do not use a hyphen "-" to separate text, as this will not work. For example, eZ systems always prepend eZ to their extensions. Prepend a useful name before the function of the extension.

In addition to the functions available in eZ publish, there is a community effort to build extensions to offer additional functionality. You can check this out at <http://pubsvn.ez.no>.

Adding an Extension

To add a new extension:

- Decide upon the kind of extension to be created, based upon the previous discussion.
- Create a new directory with the name of your new extension. eZ publish expects extensions to be located within the extension directory of the eZ publish root directory.
- Inform eZ publish about your new extension by adding it to the configuration file: `site.ini`.

For example, to activate the `newExtension` extension, add the following to the main configuration file (`settings/site.ini`):

```
File: settings/site.ini  
[ExtensionSettings]  
ActiveExtensions[]=newExtension
```

If the extension is to be used for a specific site design, a different `site.ini` must be used instead. For example, the standard site design for users is named `user`, but as of eZ publish 3.2, there are additional designs such as `news`, `intranet`, and so on. If you wish your extension to be present in one of these site designs but not the others, add the extension to the configuration file for that particular design:

File: `settings/siteaccess/<mydesign>/site.ini.append`

```
[ExtensionSettings]
ActiveAccessExtensions[]=newExtension
```

Here `<mydesign>` is the specific site design where your extension is to be used.

Locating Your Extension

Each extension requires its own subdirectory within the extension directory. The extension called `newExtension` will be located within the `extension/newExtension` subdirectory.

Within this new directory, eZ publish expects to find a settings file that describes what type of extension it is. Depending upon this type, other directories are expected to exist. The following table lists the standard set of subdirectories that can be recognized and used by an extension:

Directory	Description
<code>actions/</code>	New actions for forms
<code>datatypes/</code>	New datatypes to enhance a class
<code>eventtypes/</code>	New event types for workflows
<code>modules/</code>	New kernel modules and views
<code>settings/</code>	Settings for this extension
<code>translations/</code>	New translation extensions
<code>design/</code>	New design extensions

There is no need to include all these directories in your extension. If, for example, your extension is a new datatype, then only the `settings`, `design`, and `datatypes` directories need be included.

The settings directory is always present, as it contains files used to inform eZ of the nature of the extension. The only exception is with template operators, as discussed later in this chapter.

Example Directory Extensions

The following examples list the files and subdirectories associated with a typical extension directory. Notice that the subdirectories within each extension area follow the same structure as those within the eZ publish kernel and design areas.

Later in this chapter example, the content of these files will be given.

Depending on the type of the extension, you would typically have the subdirectories listed in the following sections inside your `extension/newExtension/` folder:

Datatype Extension

A datatype extension contains settings that notify eZ publish that there is a new datatype (specified in the `content.ini`) and that the extension design directory should be used (specifically, the `design.ini` file).

Here is a list of files typically affected by the creation of a datatype extension:

```
settings/content.ini.append
settings/design.ini.append
datatypes/newDatatype/newDatatype.php
design/standard/templates/content/datatypes/edit/newDatatype/newDatatype.tpl
design/standard/templates/content/datatypes/view/newDatatype/newDatatype.tpl
```

Module Extension

For a very simple module with only a single view and no functions, the extension directory would typically contain the following files:

```
settings/module.ini.append
settings/design.ini.append
modules/newModule/module.php
modules/newModule/newView.php
design/standard/templates/newView.tpl
```

Workflow Eventtype Extension

The following workflow is again very simple, with a single new eventtype and template to show that view.

```
settings/workflow.ini.append
settings/design.ini.append
eventtypes/event/myEvent/myEventtype.php
design/standard/templates/workflow/eventtype/result/event_myEvent.tpl
```

These examples may be used either as separate extensions or within the same extension, depending on your design needs. It may be the case that your extension is large and requires the addition of new datatypes, workflow events, modules, and other additions. If you use separate extensions, ensure that each is named appropriately.

The directories discussed here are automatically recognized by eZ publish, but others can be added for your own use. For example, documentation should be handled in the same way and given its own documentation subdirectory.

In some cases, it is good practice to follow the directory structure eZ publish employs, for example, when files such as database updates are added. Here files containing the SQL update code are specified as follows, with a second file describing what the update file does.

```
update/database/mysql/3.2/dbupdate-3.1-1-to-3.2-1.sql
update/database/mysql/3.2/dbupdate-3.1-1-to-3.2-1.info
```

Documentation on Extensions

Popular coding languages use documentation generators to create human-readable documents, based on the code structure and comments entered by developers. PHP is no exception to this, and there are several documentation generators available. eZ systems have chosen to design their comments to the format that doxygen uses. Doxygen can be found at <http://www.doxygen.org>. If you have this installed on your system, you can build your documentation by running the following command from the eZ publish system directory.

```
>doxygen doc/doxygen/Doxyfile
```

This allows the documentation to be referenced from within the eZ publish administration site through the URI <http://<your-admin>/reference/view/ez>.

This documentation provides very useful low-level details of PHP files, linked together to allow navigation.

For a high-level view of the eZ publish system, the eZ publish documentation is located on the eZ publish website at http://ez.no/ez_publish/documentation.

Also available is a good introduction to the system, although only for versions 3.0 and 3.1, found at <http://pubsvn.ez.no/sdk/> and <http://pubsvn.ez.no/manual>.

Modules

The first type of extension we will look at is a module. Having registered the new extension as described, the next step is to inform eZ publish that the extension contains modules. This is accomplished by adding a `module.ini.append` file containing the name of the extension as shown.

```
File: extension/moduleExtension/settings/module.ini.append
[ModuleSettings]
ExtensionRepositories[]=moduleExtension
```

If the module contains templates to render output from the module, then once again, eZ publish must be told to use the templates from the design directory in the extension as shown below.

```
File: extension/moduleExtension/settings/design.ini.append
[ExtensionSettings]
DesignExtensions[]= moduleExtension
```

Module Definitions

Modules allow information to be displayed through the view mechanism. An object is normally shown by a call to `/content/view` or when edited by `/content/edit`. Here content is the module and view and edit are the views.

Each module view is defined within the module definition file, `module.php`, located within the `modules` directory underneath the module name `extension/moduleExtension/modules/newModule/module.php`.

Notice that we have declared the name of the module to be `newModule`. eZ publish automatically recognizes new modules if they have a unique name and contain a separate `module.php` file. There is no need to specify this name within a configuration file.

This module definition file contains the specification for each module view, the access permissions granted to users, which parameters are used, and other features that will be discussed in the coming sections.

Module Names and Views

To enable eZ publish to interoperate with new module extensions, the definition file must use recognized variables, such as `$Module`, `$ViewList`, and `$FunctionList`.

An array variable named `$Module` must be initialized with the name of the module:

```
$Module = array("name" => "newModule" );
```

Module views are initialized by declaring the variable `$ViewList` as an array:

```
$ViewList = array();
```

With the views initialized, a simple view can be added:

```
$ViewList["newview"] = array("script" => "newView.php" );
```

This points the view `newview` to the correct PHP file `newView.php`. The URI for this view would be `http://<your-site>/newmodule/newview`.

View Permissions

Permissions allow users to have a predefined access to a view. Once permissions are declared in the definition file, the administrator can use these within the administration interface. If no permission is set for a module, it is left up to the main site configuration file, `site.ini`, to decide whether a whole module is accessible or not. By default, the module and its views are not available to anyone but the Administrator role.

Permissions for a view can be added by first declaring the permission as a `$FunctionList` variable, and then adding this to the view definition. For instance, to add a permission called `read` to a view, the following code can be used:

```
$ViewList["newview"] = array("script" => "newView.php",
                             "functions" => array('read' ) );
$FunctionList['read'] = array();
```

In this example, the `read` permission is declared and then added as an array element to the `$ViewList["myview"]` array. Note that the order of declaration is not important here.

Another view may allow editing, and shown below are two views, one with `read` permissions, and the other allowing editing:

```
$ViewList["newview"] = array("script" => "newView.php",
                             "functions" => array('read' ));
$ViewList["neweditview"] = array("script" => "newEditView.php",
                                 "functions" => array('edit' ));
$FunctionList['read'] = array();
$FunctionList['edit'] = array();
```

Here the permissions are used separately; one for `read` and another for editing, but such permissions can be applied to a single view:

```
$ViewList["newview"] = array("script" => "newView.php",
                             "functions" => array('read', 'edit' ));
$FunctionList['read'] = array();
$FunctionList['edit'] = array();
```

It is important to note that it is up to the developer of the module and view to test the permissions that a user has and to present the appropriate response to the user request.

View Parameters

Module views are not very useful on their own unless parameters are passed in to allow adequate customization of the view response.

eZ publish expects uses to be declared for each parameter view, and performs checks to ensure that these parameters are valid for their declared type. For example:

```
$ViewList["newview"] = array("script" => "newView.php",
                             "functions" => array('read' ),
                             "params" => array("ViewMode", "NodeID"
),
                             "unordered_params" =>
                             array("language" =>
"Language",
                             "offset" => "Offset" )
);
$FunctionList['read'] = array();
```

There are two types of view parameters:

- **params:** Parameters that follow directly from the module and view in the URI and are listed in left-to-right order. For instance, if the URI was `/newmodule/newview/full/20`, then `ViewMode` will be `full` and `NodeID` would be `20`.
- **unordered_params:** The `unordered` term implies that these parameters can come in any order after the ordered params. For example, if the URI was `/newmodule/newview/full/20/offset/10/language/en`, `ViewMode` and `NodeID` are the same as before, and the value for `Offset` will be `10` and `Language` will be `en` (representing English).

Note the use of an associative array (`"offset"=>"Offset"`) for `unordered_params`. This means one form of the parameter can be used in the URI (`offset`) and another in the PHP script (`Offset`). Ordered parameters do not use associative arrays so the name of the parameter has to be the same as the one available in the module view script.

Unordered parameters may also be available to the template, after the module has been processed. Currently eZ publish returns the variable `$view_parameters` to the template, and if offsets have been used, this is available as `$view_parameters.offset`.

View Actions and Post Variables

Parameter values are passed to module views by means of HTML `POST` variables. Normal HTML is used to encode the data, and the form is submitted to the module view. When the data arrives, an **action** might occur, depending upon the input data.

The following example lists several actions that can occur:

```
$ViewList["newview"] = array("script" => "newView.php",
                             "single_post_actions" =>
                             array('ConfirmButton' => 'Confirm',
                             'CancelButton' => 'Cancel' ),
                             "post_actions" =>
array('BrowseActionName' ),
                             "post_action_parameters" =>
                             array('Confirm' =>
```



```
array('ConfirmParameter' =>
      'ConfirmValue' ));
```

There are four parts to viewing actions.

single_post_actions

A web form has a single action to perform. The eZ template will set an input tag with the desired action, and when the submit button clicked, its name will be passed through to the specified eZ module view. The submit buttons allow either a confirmation or a cancellation action. For example:

```
<form method="post" action="/newModule/newview/">
  <input type="submit" name="ConfirmButton" value="Confirm" />
  <input type="submit" name="CancelButton" value="Cancel" />
</form>
```

An associative array is used to transform the POST action name to a name that will be used by the view script. This is examined by the script as follows:

```
if ($Module->isCurrentAction('Confirm' ) )
{
  /* Do something */
}

if ($Module->isCurrentAction('Cancel' ) )
{
  /* Do something else */
}
```

post_actions

The `post_actions` variable offers flexibility with the naming of POST variables. When the system sets the action for the module it first examines whether the POST variable matches anything within `single_post_actions`. Failing that, it will look in `post_actions` to find a match.

The system will take the *name* of the `single_post_action` as the module action, but will take the *value* of the `post_actions` variable as the module action.

This allows the developer to avoid needing to explicitly declare actions within the module configuration file.

```
<form method="post" action="/newModule/newview/">
  <input class="button" type="submit" name="BrowseActionName"
        value="MyNewAction" />
</form>
```

Note that the name of the input tag matches with the previously declared `post_actions` value in the view definition. The value of the input tag is used as to set the current action and is retrieved as earlier:

```
if ($Module->isCurrentAction('MyNewAction' ) )
{
    /* Do new action */
}
```

post_action_parameters

post_action_parameters are parameters for each action. These allow the script to refine how the action will be processed.

In the above example, the post_action parameter is defined as an associative array, the current action Confirm is used as the key, and its value is defined as another associative array. The key for this second array is used as the name for the actions parameter, ConfirmParameter, and the value, ConfirmValue, is taken from the input tag, as shown by the following example:

```
<form method="post" action="/newModule/newview/">
    <input type="submit" name="ConfirmButton" value="Confirm" />
    <input type="hidden" name="ConfirmValue" value="Yes, confirm" />
</form>
```

This happens when processing the following code:

```
if ($Module->isCurrentAction('Confirm' ) )
{
    if ($Module->hasActionParameter('ConfirmParameter' ) )
    {
        $test = $Module->actionParameter('ConfirmParameter' );
    }
}
```

In this example, the value of \$test will be "Yes, confirm".

Normal Post Variables

Lastly, ordinary POST values can be checked for and used within the script without any mention in the definition of the view. If the HTML code is as follows:

```
<form method="post" action="/newmodule/newview/">
    <input type="submit" name="NewTestButton" value="Push me" />
</form>
```

the script could check for this POST variable as follows:

```
include_once('lib/ezutils/classes/ezhttpptool.php' );
$http =& eZHTTPTool::instance();
if ($http->hasPostVariable('NewTestButton' ) )
    $testButtonValue = $http->postVariable('NewTestButton' );
```

Here the eZ HTTP library is invoked to retrieve the POST variable.

View Navigation

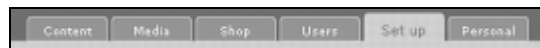
Assigning a view navigation variable is a method eZ publish employs to structure the administration interface. Each view is a member of a particular area, such as Set up or Media, and is shown in the left-hand side navigation bar of the administration interface.

This allows the interface to perform some action such as highlighting the area tab when the view is active.

To assign a navigation area to a view, set a `default_navigation_part` parameter to the view definition as shown:

```
$ViewList["newview"] = array("script" => "newView.php",
                             "default_navigation_part" =>
                             'ezsetupnavigationpart');
```

An example of this in action is shown by the following screenshot, where the Set up tab is highlighted:



Module Coding

Now that we've seen the module definitions and configuring the permissions and the parameters we will require, we will move on to look at the details of coding the module.

Reading Module Input

In the previous section, the module definition file was shown to declare the expected parameters a view takes as input. Parameters are passed to the view with the `$Params` array. For example:

```
$Module =& $Params['Module'];
$NodeID = $Params['NodeID'];
$Offset = $Params['Offset'];
```

For parameters that are unordered and therefore either present or not, it is good practice to check the input values and declare a default in case the variable is not present. For example:

```
$Offset = $Params['Offset'];
if (!is_numeric($Offset ) )
    $Offset = 0;
```

Returning Information

The eZ publish system uses a template array variable called `$module_result` to hold the output from a module view. For example, the output from the `kernel/view` module view contains the following information in an array:

module_result Array Examples	Description
content	Direct output from the module view held as a string
view_parameters	Specifies the parameters used by the module
path	Array of elements that describes the path to the current element
title_path	Similar to the path but without the root element
section_id	Section identifier of the current node
node_id	Identifier of the current node
navigation_part	The navigation area to which the view is assigned

Not every view uses all of these parameters as part of the module result. The recommended minimum is to use content and path. Others can be added depending on what your template expects. For example,

File: `extension/moduleExtension/modules/newModule/newView.php`

```
$Result = array();
$Result['content'] = "Hello there! ";
$Result['path'] = array(array('url' => false,
                             'text' => 'New Module' ),
                        array('url' => false,
                             'text' => 'New View' ) );
```

In this example, the content parameter is assigned a string value. The path is assigned a simple list containing information about the module and its view—in practice, the path will be more descriptive and would likely contain the current node_id and maybe other information such as the URL alias of the node.

Processing a Template

The advantages of using a module view become apparent when templates are used to customize the output in the view.

Before the template can be used, it must be initialized as shown:

File: `extension/moduleExtension/modules/newModule/newView.php`

```
include_once('kernel/common/template.php' );
$tpl =& templateInit();
```

If the template requires input values to be present as parameters, these can be set:

File: `extension/moduleExtension/modules/newModule/newView.php`

```
$tpl->setVariable('my_parameter', $myParameter );
$tpl->setVariable('my_other_parameter', $myOtherParameter );
```

This assigns the PHP variables `$myParameter` and `$myOtherParameter` to the `my_parameter` and `my_other_parameter` template variables respectively. The template code `{ $my_parameter }` and `{ $my_other_parameter }` will retrieve the values as expected.

Assigning the value of `$Result['content']` invokes the processing of the template, as shown:

```
$Result = array();
$Result['content'] = $tpl->fetch("design:newmodule/newview.tpl");
```

In this example, the template is retrieved from the appropriate design folder and processed by the call to `$tpl->fetch()`. The content element will contain a string representing the output from the processed template.

Redirecting a Module

Some modules and views act as wrappers for other modules and views. For example, within the content module, there is the action view that redirects input according to the value of POST variables.

eZ publish provides several routines to aid with redirection.

redirectTo

The simplest `redirectTo()` method takes a path as its parameter. This allows you to move the processing from your view to another module, which will provide output in a predetermined way. For example, you can redirect to the home page if something irregular occurs, or as in the following example:

```
$Module->redirectTo('/content/view/full/2/' );
```

redirect

A common use for `redirect()` is to invoke the processing abilities of another module and view given parameters that have been determined from the current view. For example, here the code invokes the `edit` view of the content module, with the object ID, its current version, and the currently used language:

```
$Module->redirect('content', 'edit', array($contentObjectID, $version,
                                           $language) );
```

redirectToView

This method is a convenient method that assumes the view parameter to be within the current module, with the final array parameter the same as before:

```
$Module->redirectToView('edit', array($contentObjectID, $version,
                                     $language) );
```

redirectionURI

In some cases, you may need to add a parameter to the location in the `redirectTo()` method, but first of all you will need the basic URI. This is where `redirectionURI()` will help you:

```
$Module->redirectionURI('content', 'edit', array($contentObjectID,
                                                $version,$language) );
```

This method only creates a valid redirection URI, and does not perform the redirection.

Module Functions

Functions invoked by the template author use the `fetch` command. The aim is to perform a check of some kind either from the present environment or to find values from the database. eZ publish provides a large number of utility functions to make the task of a function author relatively simple.

The remainder of this section will discuss how to create a function by showing how to build one that returns information for the following example:

```
{let
my_function_output=fetch('newmodule','newfunction',hash(number,999))}
Output from function is: {$my_function_output}
{/let}
```

The function performs a simple calculation to compute the square of 999.

Registering a Function

Functions must be defined in a definition file called `function_definition.php` and located in the same directory as `module.php`.

```
File:
extension/moduleExtension/modules/newModule/function_definition.php

$FunctionList = array();
$FunctionList['newfunction'] = array('name' => 'newfunction',
                                     'operation_types' =>
                                     array('read' ),

                                     'call_method' =>
                                     array('include_file'
=>'extension/moduleExtension/modules/newModule/newModuleFunctionCollection.php',
                                     'class' =>
                                     'newModuleFunctionCollection',
                                     'method' => 'fetchNumber' ),
                                     'parameter_type' => 'standard',
                                     'parameters' => array(array(
                                                         'name' => 'number',
                                                         'type' => 'integer',
                                                         'required' => true )
                                                         ) );
```

The name of your function should be the same as the key to the PHP array:

```
$FunctionList['newfunction'] = array('name' => 'newfunction',
```

The purpose of the `operation_types` function is to allow access to the function:

```
'operation_types' => array('read' ),
```

The call method informs eZ of your script location, the class name, and the method inside the class to be used when the function is invoked:

```
'call_method' => array('include_file' =>
                        'extension/moduleExtension/modules/
newModule/newModuleFunctionCollection.php',
                        'class' => 'NewModuleFunctionCollection',
                        'method' => 'fetchNumber' )
```

In this example, the PHP file holding the function is kept within the module directory, which is standard practice.

Finally, the function parameters must be defined. The type of parameter should be set as standard. If you use different types, the type should be mixed.

At the moment, eZ publish does not check the type of function parameters. This may change in the future.

```
'parameter_type' => 'standard',
'parameters' => array(
    array('name' => 'number',
          'type' => 'integer',
          'required' => true ) ) );
```

In this example, there is a single integer type parameter called `number` and it must be included when the function is included. If it is not included, eZ publish will display an error message.

Coding Functions

With the function defined and registered with eZ publish, all that remains to be written is the function code itself:

File: `extension/moduleextension/modules/newmodule/`

newmodulefunctioncollection.php

```
class NewModuleFunctionCollection
{
    function NewModuleFunctionCollection() {}
    function &fetchNumber($object_id )
    {
        return array('result' => 'Square of found number '.$number.'
                        is '.$number*$number);
    }
}
```

Datatypes

The default datatypes with eZ publish 3.2 are comprehensive and sufficient for most sites. When there is need for a new datatype, however, it is relatively easy to include.

The benefits of creating a new datatype balance the cost for its design, implementation, and testing. The aim is to make the management of objects easier and less time-consuming. If an existing datatype works but is difficult for the user to work with, it is worthwhile thinking about a new datatype to resolve the issue.

In this section, we will show how to create new datatypes. To do this, we need to inherit from the `eZDataType.php` base class and override a number of its functions, which allows the datatype to perform our new actions when objects are viewed and edited.

In the next chapter, a detailed example of a complex datatype will be presented.

Datatype Settings

The datatype configuration file, `content.ini.append`, is where eZ is informed about the datatype.

File: `extension/datatypeExtension/settings/content.ini.append`

```
[DataTypeSettings]
ExtensionDirectories[]=datatypeExtension
AvailableDataTypes[]=newDatatype
```

A difference here is the use of the `AvailableDataTypes[]` array. In addition to letting eZ publish know about the datatype within the extension, the name of the datatype must be explicitly declared.

As usual there will be designs for the datatype, and eZ publish must know of the design directory:

File: `extension/datatypeExtension/settings/design.ini.append`

```
[ExtensionSettings]
DesignExtensions[]=datatypeExtension
```

Datatype Templates

When designing a new class, an attribute is created from a datatype representation. To render this attribute within the class edit view, a template is required. Similarly when an object is being viewed or edited, there must be a template for the attribute/datatype for it to be shown.

For most situations a datatype can be declared within a class, with only its name and identifier required to be entered—eZno—if it is allowed, in case the developer forgets to fill in a value for each content object created from it.

If a datatype requires options to be set that influence the choices that the developer will have to make when creating the content object, an extra template is required. For example, the `ezselection` datatype requires the developer to define the enumeration to be used for the content object, and whether the selection will be a multiple choice or single selection.

A special class edit template for the `newDatatype` datatype, which would likely contain choices for the developer to select, can be found at:

```
extension/datatypeExtension/design/standard/templates/class/
datatype/edit/newDatatype.tpl
```

Once the class has been created, two different templates allow the datatype within that new content object to be both viewed and edited:

```
extension/datatypeExtension/design/standard/templates/
content/datatype/edit/newDatatype.tpl
extension/datatypeExtension/design/standard/templates/
content/datatype/view/newDatatype.tpl
```

The Datatype Wizard

The datatype wizard allows the developer to create a generic PHP file that can be further customized by the developer to create the datatype. This wizard is part of the RAD tools found within the Set up section of the administration site.

The datatype wizard is found at <http://<admin>/setup/datatype> and consists of the following steps:

1. The Introduction to datatypes welcome screen.
2. You are asked for the name of the datatype. This should be something suitable as it will be used in your templates. If your datatype requires further class edits, like the `ezselection` example described before, tick the checkbox in Settings. This will allow for input at the class level instead of the normal object level.

Template operator wizard

Datatype wizard

Basic information

Name of datatype
NewDatatype

Descriptive name of datatype
A test datatype

Settings

☒ Handle input on class level

Next >> Restart

- From your responses, the information in the next screen is prefilled. Note that your datatype name is now part of the PHP class name. Make sure you do not put spaces in your operator, otherwise PHP will complain.
- The following screenshot displays the remaining questions. Once finished, click the Download button to save your datatype:

Template operator wizard

Datatype wizard

Optional information

Name of class
eznew_datatype

Constant name
EZ_DATATYPESTRING_NEW_DATATYPE

The creator of the datatype
Administrator User

Description of your datatype

The first line will be used as the brief description and the rest are operator documentation.
Handles the datatype new_datatype
By using new_datatype you can ...

Once the download button is clicked the code will be generated and the browser will ask you to store the generated file.

Download >> Restart

This information helps prefill the comment text within the PHP file, but does not help with coding the datatype. Hence, the user will need to pursue this further. The wizard is used for creating the generic datatype file `ez<name>type.php`, where `<name>` is the name given to the datatype. Read on for details on how to make a useful datatype.

Implementing the Datatype

Each datatype inherits from the base class `eZDataType`. The file name of the new datatype must end with `type.php`, for example, `newdatatype.php`, for the system to use the file. This file can be created manually or by using the datatype wizard:

`extension/datatypeExtension/datatypes/newDatatype/newDatatype.php`

Constructing a Datatype

The general skeleton code for a datatype is as follows:

```
<?php
include_once("kernel/classes/ezdatatype.php" );
define("EZ_DATATYPESTRING_NEWDATATYPE", "newdatatype" );

class newDataTypeType extends eZDataType
{
    function newDataTypeType()
    {
        $this->eZDataType(EZ_DATATYPESTRING_NEWDATATYPE, "None" );
    }
}

eZDataType::register(EZ_DATATYPESTRING_NEWDATATYPE, "newdatatype"
);
?>
```

The `newDataTypeType()` constructor registers with the base class by passing the name of the datatype to it. Notice that the datatype is registered with the system at the end of the file using the same name.

```
eZDataType::register(EZ_DATATYPESTRING_NEWDATATYPE, "newdatatype"
);
```

Storing Datatype Information

For each datatype within the eZ publish system there are two types of attributes. Class attributes store information about which class they are part of, the name of the attribute, any default values the object attribute should have, and other custom information the object attribute may find useful. Object attributes use the information from the class attributes to store run-time information about the attribute. Any number of object attributes can be created from the class attributes.

The attributes of an object examine the fields within the SQL database for its class attributes during the creation of the object. For example, the class attribute for the `ezstring` datatype stores default information within the SQL column `data_text1`, which is read by the object attribute for `ezstring`, and stored within the SQL column `data_text`. Similarly, a class attribute for `ezinteger` stores default information within a column named `data_int1`, and the object attribute stores the result within `data_int`. All

datatypes work in the same way, and some may use more database fields to store information.

It is worth examining the PHP class files responsible for class and object attributes to better understand the relationship they have with the database. With this understanding in place, it becomes easier to create complex datatypes. These files are:

```
kernel/classes/ezcontentclassattribute.php
kernel/classes/ezcontentobjectattribute.php
```

If you investigate existing datatypes, there are good examples of other SQL column values being used in various situations.

Initializing with Default Values

For most datatypes, it is possible to assign a default value to be used when an object is created from the class. In the absence of a user-defined default value, the developer can make suitable provisions from the developing stage itself.

There are two function interfaces the developer can use to assign a default value.

Class Default Values

This interface allows the class attribute for the datatype to be assigned a value, which can then be changed by the user and stored. For example:

```
function initializeClassAttribute(&$classAttribute )
{
    if ( $classAttribute->attribute('data_int1' ) == null )
    {
        $classAttribute->setAttribute('data_int1', 10 );
    }
    $classAttribute->store();
}
```

Here the `data_int1` parameter refers to a database column from the `ezcontentclass_attribute` database table. It is queried to find whether it has a value assigned to it; if not, a new value is stored. This function is used when the class is edited and there is no value present. Depending upon the datatype, other SQL columns may be used to store a default value, for example, `data_float1` for floating-point numbers or `data_text1` for textual information.

Object Default Values

The second interface applies to the object attribute. It examines whether there is a default value assigned to the class attribute, and uses it to set the default for the content object.

```
function initializeObjectAttribute(&$contentObjectAttribute,
                                $currentVersion,
                                &$originalContentObjectAttribute
                                )
{
    if ( $currentVersion != false )
```

```

        {
            $dataInt = $originalContentObjectAttribute-
>attribute("data_int" );
            $contentObjectAttribute->setAttribute("data_int", $dataInt
);
        }
        else
        {
            $contentClassAttribute =& $contentObjectAttribute->
                contentClassAttribute();
            $default = $contentClassAttribute->attribute("data_int1" );
            if ($default != 0 )
            {
                $contentObjectAttribute->setAttribute("data_int", $default
);
            }
        }
    }
}

```

With this example the concept of versions is used when initializing the object. If the object being edited is at its current version then the default value used is the same as the previous version. In the case of a new version, where the object is not at the current version, then the default value from the class attribute is used instead (data_int1).

Working with Class Attributes

When the user wishes to alter the default value into something more suitable, the following functions are used.

Reading Input Values

The `fetchClassAttributeHTTPInput()` function retrieves the values of the variable from the POST variable that was submitted when the user stored the class.

```

function fetchClassAttributeHTTPInput(&$http, $base, &$classAttribute
)
{
    $defaultName = $base."_NewDatatype_". $classAttribute-
>attribute('id');
    if ($http->hasPostVariable($defaultName ) )
    {
        $defaultValue = $http->postVariable($defaultName );
        if ($defaultValue == "")
        {
            $defaultValue = "0";
        }
        $classAttribute->setAttribute("data_int", $defaultValue );
    }
    return true;
}

```

The variable `$defaultName` is constructed using information from the class attribute based on the knowledge that it is dealing with a class attribute.

Input tags within web forms must be assigned names that eZ publish can recognize, and a typical example of such a tag would be:

```
<input type="text" name="ContentClass_NewDatatype_20" value=""
size="8"
maxlength="20" />
```

The `$base` parameter, when used for class attributes, is always `ContentClass`. Hence, `$defaultName` will be a valid POST variable than can be used to retrieve the value.

This method also stores the value of the POST variable. Normally, this is performed by the `storeClassAttribute()` function, but almost all current eZ publish datatypes store the value after it has been fetched from the web form.

Validating Datatype Input

Input that will be stored within the database should be validated before it is entered. The `validateClassAttributeHTTPInput()` function performs this task and returns a value indicating the result of the validation. In the following example, only the values "0" and "1" are acceptable, with "1" being a special case—any other value is invalid.

```
function validateClassAttributeHTTPInput(&$http,
                                         $base,
                                         &$classAttribute )
{
    $defaultName = $base . "_NewDatatype_" .
                   $classAttribute->attribute('id' );
    if ( $http->hasPostVariable($defaultName ) )
    {
        $postValue = $http->postVariable($defaultName );

        if (is_numeric($postValue)
        {
            if ($postValue == "0" )
            {
                return EZ_INPUT_VALIDATOR_STATE_ACCEPTED;
            }
            else if($postValue == "1" )
            {
                return EZ_INPUT_VALIDATOR_STATE_INTERMEDIATE;
            }
        }
    }

    return EZ_INPUT_VALIDATOR_STATE_INVALID;
}
```

If the result is invalid, the user is informed that the input was wrong, and if it is acceptable, the input is stored. If, however, the returned value was `EZ_INPUT_VALIDATOR_STATE_INTERMEDIATE` then the `fixupClassAttributeHTTPInput()` function is called to repair the value.

Depending upon the nature of the datatype, a value can be reset to another. For instance, if boundary conditions are not met, the value may be moved to the nearest boundary. This

feature exists to help the user make the right choice and ensure data integrity within the database.

In the following example, the fix routine will set the POST variable to 0 if the input value was 1.

```
function fixupClassAttributeHTTPInput(&$http, $base, &$classAttribute)
{
    $defaultName = $base . "_NewDatatype_" .
                    $classAttribute->attribute('id' );
    if ($http->hasPostVariable($defaultName ) )
    {
        $defaultValue = $http->postVariable($defaultName );
        if ($defaultValue == "1" )
        {
            $http->setPostVariable($defaultName, "0");
        }
    }
}
```

Working with Object Attributes

Object attributes work in a very similar manner to class attributes and therefore follow the same reasoning. Instead of repeating examples that are almost identical to class attribute examples, the object attribute APIs are presented for use in your own functions:

```
function validateObjectAttributeHTTPInput(&$http, $base,
                                          &$contentObjectAttribute)

function fetchObjectAttributeHTTPInput(&$http, $base,
                                       &$contentObjectAttribute)

function storeObjectAttribute(&$contentObjectAttribute)

function fixupObjectAttributeHTTPInput(&$http, $base,
                                       &$contentObjectAttribute)
```

The \$base variable is defined as ContentObjectAttribute within the context of content object attributes.

One of the main differences between class and object attributes is in the use of database columns. With class attributes, there is a wide range of columns the user can use to store various defaults and current states, whereas the content object attributes columns store the values that will be displayed as part of a web page.

Other Datatype Functions

Apart from working with class and object attributes, the datatype can implement other API functions that inform the system what this datatype is able to do.

- `isIndexable()`: Returns a boolean value to indicate whether the values of this datatype should be used as search data:

```
function isIndexable()
{
    return true;
}
```

- `isInformationCollector()`: Returns a boolean value to indicate that the datatype should be used as an information collector. If the boolean returns true, this adds a checkbox in the administration class edit view to allow the user to choose whether the datatype acts as an information collector. A value of false prevents the checkbox from being shown.

```
function isInformationCollector()
{
    return true;
}
```

- `title()`: Selects which part of the datatype to use for the title of the datatype. This gives the datatype its name.

```
function title(&$contentObjectAttribute )
{
    return $contentObjectAttribute->attribute('data_text' );
}
```

- `metaData()`: Selects which text to use when storing search data.

```
function metaData(&$contentObjectAttribute )
{
    return $contentObjectAttribute->attribute('data_text' );
}
```

Template Design

Earlier it was stated that input tags within web forms need the correct name for the datatype, otherwise the system will not recognize the input. Within the context of a class or object attribute, it is possible to define the name of the input tag in a dynamic manner.

```
extension/datatypeExtension/design/standard/templates/class/datatype/
edit/newDatatype.tpl
```

```
{default name=concat("ContentClass_NewDatatype_",
$class_attribute.id)
    value=$class_attribute.data_text1}
<input type="text" name="{ $name}" value="{ $value|wash}" size="30"
maxlength="60" />
{/default}
```

Class edit templates have the variable `$class_attribute` available:

```
extension/datatypeExtension/design/standard/templates/content/datatype/
edit/newDatatype.tpl
```

```
{default
name=concat("ContentObjectAttribute_NewDatatype_data_text_",
    $attribute.id) value=$attribute.data_text}
```



```

        <input type="text" name="{ $name}" value="{ $value|wash(xhtml)}"
        size="70" />

    {/default}

```

Content object attributes, like class attributes, have the variable `$attribute` available to templates:

```

extension/datatypeExtension/design/standard/templates/content/datatype
e/view/newDatatype.tpl
{ $attribute.data_text|wash(xhtml)}

```

This view example is very simple and displays only the textual string from the database value. This value is *washed* before being rendered in case the string contains strange values for the browser.

Complex Datatypes

Some datatypes, such as the object relation list and XML data field, are complex and require further explanation. All complex datatypes also use the datatype API to provide the connection with the system, but beyond this, they use complex data structures to achieve their purpose.

In the next chapter, there will be an example of a complex datatype used for an e-commerce system, the category datatype.

Template Operators

Template operators act like program functions; they take parameters and return a single value. Thankfully, that single value may be an array, which allows you to return more than one value!

Template operators can be handled like ordinary extensions, but in fact are not treated like other extensions by eZ. There is no specific directory that eZ looks for when a template directory is declared. However, this section will present examples of using template operators within the extension environment to maintain consistency.

Adding a PHP Command

As of eZ publish 3.2, the ability to assign a simple PHP command as a template operator has been made available. In most cases, this allows the use of a PHP command through to the template level. It is activated by the declaration of the operator within the `template.ini` configuration file.

```

File: settings/template.ini

[PHP]
PHPOperatorList[]

```

```
PHPOperatorList[upcase]=strtoupper
```

In this example, the PHP function `strtoupper()` is referenced using the `upcase()` template operator. To add your own command, a new item must be added to the `PHPOperatorList` configuration array. For example:

```
PHPOperatorList[downcase]=strtolower
```

When active, such operators work as normal; for example, `{"my string"|upcase}` produces `"MY STRING"`.

There is one limitation to remember with eZ publish 3.2: the operator will not work as expected with more than one parameter. No parameters or just one will be fine; others will be ignored. You must therefore be careful when selecting the PHP operator you wish to use. If you require more than one parameter in your operator then keep reading and learn how to build your own.

The Template Operator Wizard

The template operator wizard allows you to create a generic PHP file that can be further customized to create the required operator. This wizard is part of the RAD tools found within the Set up section of the administration site, at <http://<admin>/setup/templateoperator>.

The operation of the template operator wizard is the same as for the datatype.

1. A welcome screen provides an introduction to operator templates.
2. You are asked for the name of the operator. This should be something intuitive, for it will be used in your templates. You are also asked whether the operator should handle input, provide output, and handle parameters:

Template operator wizard

Template operator wizard

Basic information

Name of operator
NewOperator

Settings

☒ Handles operator input
☒ Generates operator output

Parameter handling

No parameters

Next >> Restart

- Based on your responses in the previous screen, some information in this screen is pre-filled. Note that your operator name is now part of the PHP class name. Make sure you do not put spaces in your operator, otherwise PHP will complain.

Template operator wizard

Template operator wizard

Optional information

Name of class

TemplateNew_operatorOperator

The creator of the operator

Administrator User

Description of your operator

The first line will be used as the brief description and the rest are operator documentation.

Handles template operator new_operator
By using new_operator you can ...

Example code

If you wish you can add some example code to explain how your operator should work. The default code was made from the basic parameters you chose.

{ \$value | new_operator | wash }

Once the download button is clicked the code will be generated and the browser will ask you to store the generated file.

Download >> Restart

- Once finished, click the Download button to save your PHP file.

The wizard creates a file similar in function to the one created by the datatype wizard. When the download button is clicked, the generic operator file (template<name>operator.php) is generated, where <name> is the name given to the operator. As earlier, further coding of the operator including its setup, is left to the user. Essentially, the wizard only creates the generic template operator file, and even with the help of the operator wizard, much of the hard work for building an operator must still be carried out.

Writing an Operator

Writing your own operator gives the freedom to implement any functionality you need. As mentioned earlier, operators act like program functions by accepting parameters and returning a single value.

Registering the Operator

The `site.ini` file contains a setting that informs eZ about the location of template operators. When the system looks for operators, it first looks in system directories for a file called `eztemplateautoload.php`. This file contains the names of the operators that can be used. For example:

File: `settings/site.ini`

```
[TemplateSettings]
AutoloadPath=lib/eztemplate/classes/kernel/common/
```

It is good coding practice not to add operators to existing files, and instead to create a new `eztemplateautoload.php` file within the extension area.

To activate this, the configuration file should be modified to include the new path, as follows:

`settings/site.ini`

```
[TemplateSettings]
AutoloadPath=lib/eztemplate/classes/kernel/common/extension/operatorE
xtension/
```

eZ now expects the file `extension/operatorExtension/eztemplateautoload.php` to exist. Create this file and include within it the following code:

File: `extension/operatorExtension/eztemplateautoload.php`

```
<?php

    $eZTemplateOperatorArray = array();

    $eZTemplateOperatorArray[] = array('script' =>
'extension/operatorExtension/randomtemplateoperator.php',
    'class' =>
'RandomTemplateOperator',
    'operator_names' =>
array('randomtemplateoperator' ) );

?>
```

This code declares a new template operator named `randomtemplateoperator`. The `RandomTemplateOperator` class within the file is used when a template invokes the operator.

Coding the Operator

Template operators are far simpler than other PHP classes described before. The class does not need to inherit from any other class, and the constructor may remain empty.

```
class RandomTemplateOperator
{
    function RandomTemplateOperator()
    {
    }
}
```

Initializing the Operator

In order for eZ publish to register the operator, it must be declared. This is accomplished by the `operatorList()` function:

```
function &operatorList()
{
    return array('randomtemplateoperator' );
}
```

The `namedParameterList()` function declares the parameters for the operator. Each parameter has a type and can be optional. If the parameter is required and not passed to the operator, eZ publish will log the problem.

For our template operator, this function is as follows:

```
function namedParameterList()
{
    return array('first_param' => array('type' => 'string',
                                        'required' => false,
                                        'default' => 'default
    text')));
}
```

This declares the parameter as `first_param` and defines it as an optional string with a default value.

Executing the Operator

The operator is executed by the `modify()` function using the information supplied to it:

```
function modify(&$tpl, &$operatorName, &$operatorParameters,
               &$rootNamespace,
               &$currentNamespace, &$operatorValue,
               &$namedParameters)
{
    $firstParam = $namedParameters['first_param'];
    switch ($operatorName )
    {
        case 'randomtemplateoperator':
        {
            $operatorValue=rand(0, $firstParam );
        }
    }
}
```

```
        }  
        break;  
    }  
}
```

In this instance, the simple `rand()` function is used along with the parameter, and the `$operatorValue` variable is used to store the result. This is a simple example, but you can use any PHP function to produce the results that you need.

It must be noted that there is no explicit return value because `$operatorValue` is passed by reference, and any change to it will still exist once the function completes execution.

Workflow Events and Triggers

Every CMS system requires a workflow engine to allow users to collaborate effectively. eZ publish includes a workflow system with a selection of default workflow events and triggers to activate events.

Common workflows include publishing and unpublishing articles. If, for example, an author writes a document and publishes it, the workflow system will not publish it until the editor has approved it. The editor would view the results from the template `event_ezapprove.tpl` and possibly approve the document. If approved, the event would perform the action of publishing the document. This section discusses how to create and use your own workflow events and triggers.

Workflow Settings

As with all extensions, the configuration file for workflows must declare to the system that there is a new workflow within the extension.

File: `extension/workflowextension/settings/workflow.ini.append`

```
[EventSettings]  
ExtensionDirectories[]= workflowExtension  
AvailableEventTypes[]=event_newEvent
```

Included here is the declaration of the `newevent` event, which will be located within:

`extension/workflowextension/eventtypes/event/newevent/`

If the workflow requires a template, then the design directory must also be declared:

File: `extension/workflowextension/settings/design.ini.append`

```
[ExtensionSettings]  
DesignExtensions[]=workflowExtension
```

As with datatypes, the name of the event is used to find the appropriate PHP file. Each event name must end with `type.php`, for example:
`extension/workflowextension/eventtypes/event/newevent/neweventtype.php`
 Template directories for workflow events are located in a different place. The `result` directory is used instead of the regular `view` directory. With this example, the template path would be:
`extension/workflowextension/design/standard/templates/workflow/eventtype/result/event_newevent.tpl`

Workflow Events

The following example code contains the information required to create a basic event class:

File:
`extension/workflowextension/eventtypes/event/newevent/neweventtype.php`

```

define("EZ_WORKFLOW_TYPE_NEWEVENT", "newevent" );
class newEventType extends eZWorkflowEventType
{
    function newEventType()
    {
        $this->eZWorkflowEventType(EZ_WORKFLOW_TYPE_NEWEVENT, "None");
    }
    function execute(&$process, &$event ) {}
}
eZWorkflowEventType::registerType(EZ_WORKFLOW_TYPE_NEWEVENT,
"neweventtype");

```

Each event type must inherit from the `eZWorkflowEventType` workflow parent class.

The constructor of the event has the same name as the class and calls the constructor of the parent class by its name.

At the end of the event, it is registered with the eZ system as follows:
`eZWorkflowEventType::registerType(EZ_WORKFLOW_TYPE_NEWEVENT, "neweventtype");`

The workflow event must implement the `execute()` function, which will decide what to do about the workflow depending on its current state. For example:

```

function execute(&$process, &$event )
{
    if ( $http->hasPostVariable("FINISHED_WORKFLOW" ) )
    {
        return EZ_WORKFLOW_TYPE_STATUS_ACCEPTED;
    }
    else
    {

```

```
$process->Template = array('templateName' => 'design:workflow/
eventtype/result/event_newevent.tpl',
                           'templateVars' =>
array('request_uri' =>
eZSys::requestUri() ) );
    return EZ_WORKFLOW_TYPE_STATUS_FETCH_TEMPLATE_REPEAT;
}
}
```

The workflow in this example expects this function to return a value reflecting the state of the workflow. If the workflow is finished, then the return value of `EZ_WORKFLOW_TYPE_STATUS_ACCEPTED` should be used. Otherwise, the function loads the template named `event_newevent.tpl` and indicates that the workflow should repeat. For the user, each time the event is triggered, this event will repeat the showing of the template until an exit condition is met allowing the template to post the `FINISHED_WORKFLOW_POST` variable.

This is a very simple example, but it illustrates that the nature of an event is to examine whether this workflow is finished, or whether something else has still to occur.

The following table presents the available status commands for the workflow. When a new event type is created, it can use a single status or a combination to determine what should occur next.

Workflow Event Type Status	Status Description
<code>EZ_WORKFLOW_TYPE_STATUS_ACCEPTED</code> <code>EZ_WORKFLOW_TYPE_STATUS_WORKFLOW_DONE</code>	The workflow event has finished.
<code>EZ_WORKFLOW_TYPE_STATUS_REJECTED</code>	The workflow is rejected.
<code>EZ_WORKFLOW_TYPE_STATUS_WORKFLOW_CANCELLED</code>	The workflow is cancelled.
<code>EZ_WORKFLOW_TYPE_STATUS_DEFERRED_TO_CRON</code> <code>EZ_WORKFLOW_TYPE_STATUS_DEFERRED_TO_CRON_REPEAT</code>	The workflow is deferred to the cron daemon.
<code>EZ_WORKFLOW_TYPE_STATUS_FETCH_TEMPLATE</code> <code>EZ_WORKFLOW_TYPE_STATUS_FETCH_TEMPLATE_REPEAT</code>	The workflow points the user to a template page and waits for the next workflow call.

Workflow Event Type Status	Status Description
EZ_WORKFLOW_TYPE_STATUS_REDIRECT EZ_WORKFLOW_TYPE_STATUS_REDIRECT_REPEAT	The workflow redirects the user to another view and waits for the next workflow call.
EZ_WORKFLOW_TYPE_STATUS_NONE	The default status, which does nothing for now.
EZ_WORKFLOW_TYPE_STATUS_RUN_SUB_EVENT	The workflow runs another event if there is one.
EZ_WORKFLOW_TYPE_STATUS_WORKFLOW_RESET	The workflow is reset for reuse.

Workflow Triggers

Workflow triggers enable an event type to run based on a user action. After the creation of a workflow (<http://<your-admin>/workflow/grouplist/>) from within the administration interface, a trigger should be assigned to it (<http://<your-admin>/trigger/list/>).

By default there are four triggers that come with eZ publish, each containing a before and after connect type. A connect type is a method that allows a workflow to be associated with a function. The following figure presents this default list as shown by the administration interface:

Trigger list			
Module Name	Function Name	Connect Type	Workflow
content	publish	before	No workflow ▾
content	publish	after	No workflow ▾
content	read	before	No workflow ▾
content	read	after	No workflow ▾
shop	confirmorder	before	No workflow ▾
shop	confirmorder	after	No workflow ▾
shop	checkout	before	No workflow ▾
shop	checkout	after	No workflow ▾

Store

For example, if you have declared your workflow to activate *before* content is published, the workflow event code will be executed before content is published, and vice versa when the connector is set to activate *after* the content is published.

Defining Triggers

An additional file named `operation_definition.php` can be added to the module directory to specify new triggers. Its composition is very similar to the file

`function_definition.php`, as shown:

```
$OperationList = array();
$OperationList['newtrigger'] =
    array('name' => 'newtrigger',
          'default_call_method' =>
              array('include_file' =>
                  'extension/triggerExtension/modules/
newModule/newmoduleoperationcollection.php',
                  'class' => 'eZTriggerExtensionOperationCollection' ),
          'parameter_type' => 'standard',
          'parameters' => array(array('name' => 'node_id',
                                     'type' => 'integer',
                                     'required' => true ) ),
          'keys' => array('node_id' ),
          'body' => array(array('type' => 'trigger',
                              'name' => 'pre_newtrigger',
                              'keys' => array('node_id' ) ),
                          array('type' => 'method',
                              'name' => 'say-hello',
                              'frequency' => 'once',
                              'method' =>
                                  'sayHello',) ) );
```

In this example, the trigger `newtrigger` is defined, and a single before connect type declared as `pre_newtrigger`. There is no after connect type, so assigning a workflow to this will not do anything.

The body item defines the order for the trigger to operate. The before connect type will activate the `pre_newtrigger` and the workflow behind it. Once the workflow completes, the methods within the body will run; in this example, this is the `sayHello()` function. If there was a `post_newtrigger` designed, it would have run the workflow assigned to the after connect type.

Functions in this example are defined within the `eztriggerextensionoperationcollection.php` file.

The specified parameters are passed to every method and trigger defined within the body. The keys are used to identify information during the workflow.

The final stage is to inform eZ publish about the new file. Unfortunately it does not pick up this information automatically, requiring an addition `workflow.ini` setting:

File: `extension/triggerextension/settings/workflow.ini.append`

```
[OperationSettings]
```

```
AvailableOperations=content_publish;content_read;shop_confirmorder;
shop_checkout;newModule_newtrigger
```

This is similar to the original `workflow.ini` but with the mention of our new trigger. eZ publish will look for the `newModule` module and then for the operations file.

For further information on this aspect of workflows, read the documentation at: <http://pubsvn.ez.no/sdk/tutorials/view/workflows/>

Actions

eZ publish relies upon the use of web forms to provide two-way communication between a user and a database. **Actions** inform the system of the user's wishes within a particular context and allow it respond appropriately. For example, there are defined actions for events such as creating, editing, publishing, deleting, adding items to shop baskets, and many more. An action extension is required when you need to provide functionality that uses a different type of action.

At its simplest, an action is a redirection mechanism. For example, if the user wishes to edit an object, the edit action will redirect the user to the edit module. Besides simplifying the use of forms, implementing an action within your extension also helps to remove coding dependencies. Your HTML will not contain hard-coded URI paths to your module, and therefore, if the code within the extension changes, you do not need to change your HTML.

The `kernel/content/action.php` kernel file defines the standard set of actions and related responses. Any additions can be made within the actions subdirectory of the extensions directory.

The new action list is registered by informing the system through the `content.ini.append.php` configuration file:

File: `extension/actionExtension/settings/content.ini.append.php`

```
[ActionSettings]
ExtensionDirectories[]=actionExtension
```

eZ publish now expects the `content_actionhandler.php` file containing the actions to exist, and it will be used for processing an action if the action is not found within `kernel/content/action.php`. The path to the content handler file is as follows:

`extension/actionExtension/actions/content_actionhandler.php`

Two example actions are:

File: `extension/actionExtension/actions/content_actionhandler.php`

```
function actionExtension_ContentActionHandler(&$module, &$http,
&$objectID )
{
    if ( $http->hasPostVariable("ActionDoSomething" ) )
    {
        echo "<p>Found ActionDoSomething</p>";
    }
}
```

```
        else if ($http->hasPostVariable("ActionDoSomethingElse" ) )
        {
            echo "<p>Found ActionDoSomethingElse</p>";
        }

        return;
    }
}
```

Both of these actions print out a simple line of text to indicate that an action is being run.

The HTML required to activate this action would be:

```
<form method="post" method="post" action={"content/action"|ezurl}>
  <input name="ActionDoSomething" type="submit" value="Do Something">
  <input name="ActionDoSomethingElse" type="submit" value="Do
Something">
  <input name="ContentObjectID" type="hidden" value="99">
</form>
```

The input tags are named after the actions we wish to use. The user can select either action to invoke the correct routine.

It must be noted that for the action extension to activate a content object, a value must be included with the POST values; without these, the extension will fail to activate.

Translations

Following on from all other extensions, translation extensions are activated by including the presence of the new translation in the `site.ini` file:

```
File: extension/translationExtension/settings/site.ini.append.php

[RegionalSettings]
TranslationExtension[]=translationExtension
```

This setting informs eZ that the `translation.ts` translation file exists within the following path:

```
extension/translationExtension/translations/
```

Note that we are assuming that the programs `ezlupdate` and `Linguist` have been used to create the `translation.ts` file.

When the user performs a translation within a template, they must specify the extension directory within which to activate translation. For example:

```
{"Translate this"|i18n("newExtension")}
```

There can only be a single translation file present for an extension.

Overriding Translations

A translation extension has an additional benefit apart from providing new translations. It is possible to override translations in the default eZ system with new translations from your extension.

To do this, you use a translation file created from an existing locale, and edit it to remove all entries apart from those you wish to override. Now use the `Linguist` program to re-translate this file. If the translation is active within your extension, your new values should now be used instead of the original values.

Notifications

eZ publish provides a configurable notification mechanism where users can be informed that an event has occurred. Notification instances could be when objects are updated or published, when workflows execute, and so on. Users can choose to receive notifications in the form of a single e-mail or as a digest of messages. Users can configure their notifications by browsing to `http://<your-site>/notifications/settings`. This screen presents the current notification settings and options to configure these.

Notification Events

The settings file `settings/notification.ini` controls which events a user can configure, as shown:

File: settings/notification.ini

```
[NotificationEventTypeSettings]
RepositoryDirectories[]=kernel/classes/notification/event/
AvailableNotificationEventTypes[]=ezpublish
AvailableNotificationEventTypes[]=ezcurrenttime
AvailableNotificationEventTypes[]=ezcollaboration
[NotificationEventHandlerSettings]
RepositoryDirectories[]=kernel/classes/notification/handler/
AvailableNotificationEventTypes[]=ezgeneraldigest
AvailableNotificationEventTypes[]=ezcollaborationnotification
AvailableNotificationEventTypes[]=ezsubtree
```

This file defines the available notification event types and the event handlers that will respond to an event.

A very simple event type is presented below:

```
define('EZ_NOTIFICATIONTYPESTRING_NEWNOTIFICATIONEVENT',
        'newnotificationevent' );
class NewNotificationEventType extends eZNotificationEventType
{
    function NewNotificationEventType()
    {
```

```
        $this->eZNotificationEventType(  
            EZ_NOTIFICATIONTYPESTRING_NEWNOTIFICATIONEVENT );  
    }  
    function initializeEvent(&$event, $params )  
    {  
        $event->setAttribute('data_int1', $params['something'] );  
    }  
    function eventContent(&$event )  
    {  
        return "hello";  
    }  
}  
eZNotificationEventType::register(  
    EZ_NOTIFICATIONTYPESTRING_NEWNOTIFICATIONEVENT,  
    'newnotificationeventtype' );
```

In this example, the event type implements two important function interfaces, `initializeEvent()` and `eventContent()`, used in other parts of the system. When the event is first created, it will be initialized with content, and when the handler runs, it will ask the event for this content. When anything causes an event, such as an object being updated, the handler will execute the `handle()` function as shown:

```
class NewNotificationEventHandler extends eZNotificationEventHandler  
{  
    ....  
    function handle(&$event )  
    {  
        if ($event->attribute('event_type_string' ) ==  
            'newnotificationevent' )  
        {  
            print "Event content is: ".$event->content();  
        }  
    }  
    ....  
}
```

Extensions can include new notifications. Shown below is an example configuration file for notifications:

```
File: extension/newextension/settings/notification.ini.append  
[NotificationEventTypeSettings]  
ExtensionDirectories[]=notificationExtension  
AvailableNotificationEventTypes[]= newnotificationevent
```

The system will now expect the event type to be located at:
`extension/notificationExtension/notificationtypes/newnotificationevent/newnotificationeventtype.php`

Unfortunately, at the time of writing, there is no extension capability for event handlers in eZ publish 3.2. The main configuration file (`notification.ini`) must be used to specify new locations for event handlers if they are present in your extension.

```
File: settings/notification.ini  
[NotificationEventHandlerSettings]  
RepositoryDirectories[]=kernel/classes/notification/handler/
```

```
RepositoryDirectories[]=extension/notificationExtension/notificationh
andlers/
AvailableNotificationEventTypes[]=newnotificationevent
```

The system will now expect the event type to be located at:
 extension/notificationExtension/notificationhandlers/newnotificationevent
 /newnotificationeventhandler.php

Adding Collaborations

A collaboration is an extension of the notification system and provides notifications from the workflow system. As shown below from the settings/collaboration.ini file, the default option is to inform authorized users when the approval notification runs:

File: settings/collaboration.ini

```
[HandlerSettings]
Extensions[]
Repositories[]=kernel/classes/collaborationhandlers
Active[]=ezapprove
```

This system works by utilizing collaboration handlers to notify subscribed users when a workflow is used. Handlers can be written for each workflow event type, but at present, the system only has one available for the ezapprove event. Extensions can incorporate their own handler for new event types by declaring the extension that the handler is part of:

File:

extension/collaborationExtension/settings/collaboration.ini.append

```
[HandlerSettings]
Extensions[]=newExtension
Active[]=newevent
```

The system will now expect the handler to be located at:
 extension/collaborationExtension/collaboration/newevent/
 neweventcollaborationhandler.php

It is left for the reader to implement their own handler.

SOAP Server

SOAP (Simple Object Access Protocol) is a protocol that enables you to communicate with other systems and data sources using HTTP and XML without having to know about the programming language or operating system. SOAP works in a client-server pair to achieve its data transfer, with the client being the requestor and the server taking the requests and passing back the requested information. The good news is that eZ publish has an implementation of both a client and server for SOAP communication. In this chapter, we will show

how you can quickly set up a SOAP server and client to demonstrate SOAP usage.

The example will do two things:

- Return a text string of our choice—in this case "example 1"
- Interact with eZ publish to return system information

To work with this example you will need a copy of the latest eZ publish installed. You need to ensure that you can call both `soapclient.php` and `soapserver.php`. If you are using virtual named hosts, you will need to set up two new URLs:

```
soapserver.example.com
soapclient.example.com
```

If you prefer to use `http://localhost` set up for this example, you will need to include the name of the PHP file you are running in the URL. We have included the `VirtualHost` setup as it is the one recommended by eZ.

This should be set up to point at `soapserver.php` and `soapclient.php` respectively. Your Apache setup for the SOAP server may look something like:

```
<VirtualHost *:80>
  <Directory /path/to/files>
    Options FollowSymLinks Indexes ExecCGI
    AllowOverride None
  </Directory>
  RewriteEngine On
  RewriteRule !\.(gif|css|jpg|png|ico|js)$
/path/to/files/soapserver.php
  DocumentRoot /path/to/files
  ServerName soapserver.example.com
</VirtualHost>
```

Once you have set up your Apache configuration, you are ready to add the code for the server in `soapserver.php`.

This first section includes routines that we need to work with in the example and creates a new SOAP server object that will wait for incoming SOAP requests:

```
<?php
// Include the eZ soap server library
include_once("lib/ezsoap/classes/ezsoapserver.php" );
include_once("lib/ezutils/classes/ezsys.php" );
// Create a new server object
$server = new eZSOAPServer();
```

We now register the function that needs to be exposed and made available. In this example, we have two, `exampleName` and `systemInfo`, that need to be registered.

```
// Register functions
$server->registerFunction("exampleName");
$server->registerFunction("systemInfo", array("infoType" => "integer"
) );
```

We now process the incoming request using `processRequest`, which takes the incoming POST variables and checks for the function that needs to be called.


```
$server->processRequest();
```

We have two functions in this example; the first is a simple return of a string:

```
// Example version name
function exampleName($noFiles )
{
    $return = "example 1";
    settype($return, "string" );
    return $return;
}
```

The second is a routine that takes the parameter passed by the SOAP client and returns the required information. In this example, the SOAP client will be requesting the operating system and hostname for the eZ publish server. Most of the example is basic PHP except the `settype()` function. This enables you to set the type of the data you are passing.

```
// System information
function systemInfo ($infoType )
{
    switch ($infoType )
    {
        case "os":
        {
            $return = eZSys::osType();
        } break;

        case "hostname":
        {
            $return = eZSys::hostname();
        } break;
    }

    settype($return, "string" );
    return $return;
}
?>
```

As of eZ publish v 3.3, the following datatypes are implemented for passing data via SOAP:

- String
- Integer
- Float
- Boolean
- base64
- Array
- SOAPStruct

Now that the server is created, we need to create the client that will communicate with the server. The client in this example is called `soapclient.php`.

The following table gives an outline of the major functions required for the SOAP client connectivity:

Code	Description
<code>eZSOAPClient()</code>	Prepares the SOAP client.
<code>eZSOAPRequest()</code>	Prepares the request object ready for parameters.
<code>addParameter()</code>	Fills the object with the request information.
<code>send()</code>	Sends the request correctly formatted in XML over HTTP to the SOAP server. The response from this request is then placed in the <code>\$response</code> variable for you to query.
<code>IsFault()</code>	Checks to see if an error occurred in the call.

As before, the first part of the code includes libraries that will be required; in this case, the SOAP client and request. The SOAP client is also created here with the location of the SOAP server we wish to call: in this case, `soapserver.example.com`.

```
<?php
// Include the eZ soap client libraries
include_once("lib/ezsoap/classes/ezsoapclient.php" );
include_once("lib/ezsoap/classes/ezsoaprequest.php" );
// Create a new client object
$client = new eZSOAPClient("soapserver.example.com", "/" );
?>
```

The first client example will call the `exampleName()` SOAP server function. This is actioned by first creating a request object that will handle the request. The request string is then placed in `$request`:

```
<h2> First SOAP example </h2>
<?php
// Create a new request to find the number of files
$request = new eZSOAPRequest("exampleName",
                             "http://soapserver.example.com");
```

The created request is then sent to the SOAP server:

```
// Send the request to the server and fetch the response
$response =& $client->send($request );
```

Now `$response` contains the response from the SOAP server, and it is checked to ensure that a fault has not occurred. If all is OK, then the response example1 is displayed:

```
// Check for SOAP fault
if ($response->isFault() )
{
    // Print the SOAP fault information
    print("SOAP fault: " . $response->faultCode(). " - " .
        $response->faultString() . " " );
}
else
{
    // If everything is ok, print the result
    print("Example name is : " . $response->value() );
}
?>
```

The second example builds on the first by passing a parameter to the SOAP server requesting information. You can change the returned information by changing the value of `$infoType` from `os` to `hostname`.

```
<h2> Second SOAP example </h2>
<?php
// Create a new request to list files
$request = new eZSOAPRequest("systemInfo",
    "http://soapserver.int.visionwt.com" );

// Add parameters
$infoType = "os";
$request->addParameter("infoType", $infoType );

// Send the request to the server and fetch the response
$response =& $client->send($request );
// Check for SOAP fault
if ($response->isFault() )
{
    // Print the SOAP fault information
    print("SOAP fault: " . $response->faultCode(). " - " .
        $response->faultString() . " " );
}
else
{
    // If everything is ok, print the result
    print("System information : " . $response->value() );
}
?>
```

To run these routines, you will need to open a browser and point it at the location of your files. If you are using virtual hostnames for both server and client, you will need to call `soapclient.example.com`.

If all has gone well, then your SOAP server and client should be able to communicate. Of course, the eZ publish libraries protect you from working with raw XML. For completeness, here are two examples that show the request from the client and the response from the server. As you can see, the libraries are doing a lot of the hard work for you.

Client XML request to server

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body xmlns:req="http://www.example.com">
    <req:exampleName />
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope><?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body xmlns:req="http://www.example.com">
    <req:systemInfo>
      <infoType xsi:type="xsd:string">os</infoType>
    </req:systemInfo>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

XML response from server

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <resp:exampleNameResponse xmlns:resp=" http://www.example.com ">
      <return xsi:type="xsd:string">example 1</return>
    </resp:exampleNameResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope><?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <resp:systemInfoResponse xmlns:resp=" http://www.example.com ">
      <return xsi:type="xsd:string">unix</return>
    </resp:systemInfoResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In this example, we have demonstrated that with a few lines of code you are able to create the SOAP client and server. You should be able to build on this to interrogate other functions within eZ publish. It is advisable to review the security arrangements of the SOAP server to ensure that the SOAP server does not provide a back door into system functions. For this reason, eZ publish has implemented a username/password system into the SOAP server.

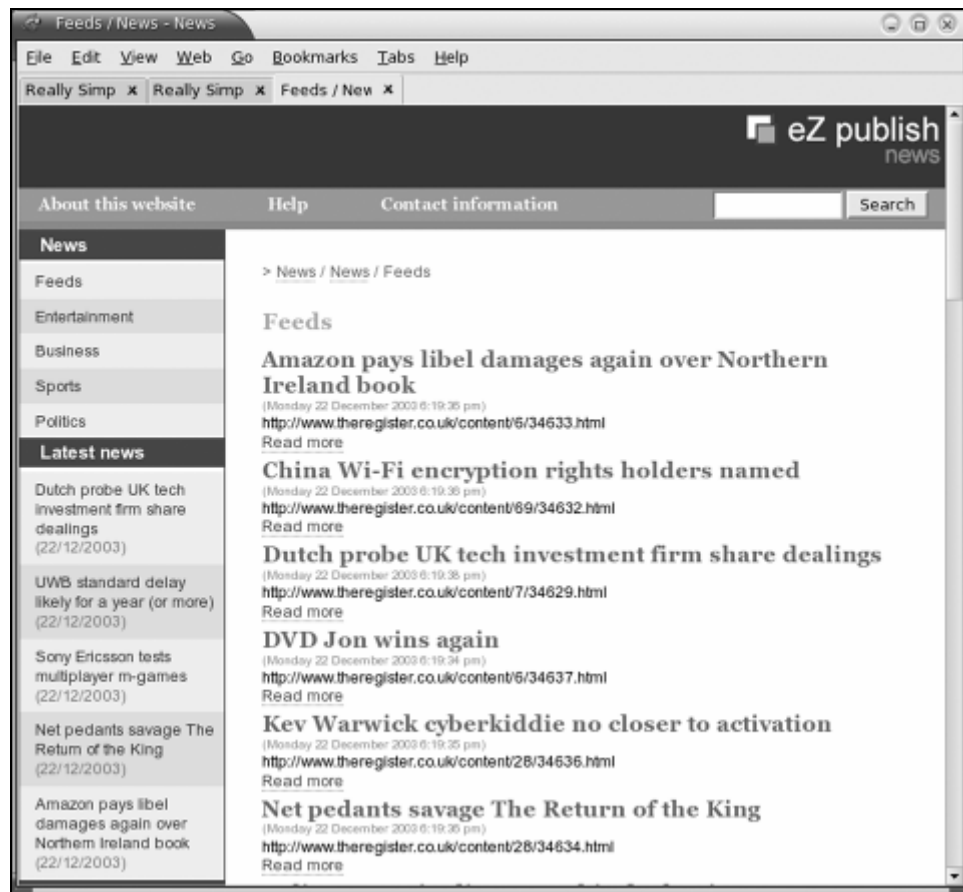
In Chapter 7, we will discuss some of the ways in which you can use SOAP to bridge to other data stores in a real-world example.

RSS (Really Simple Syndication)

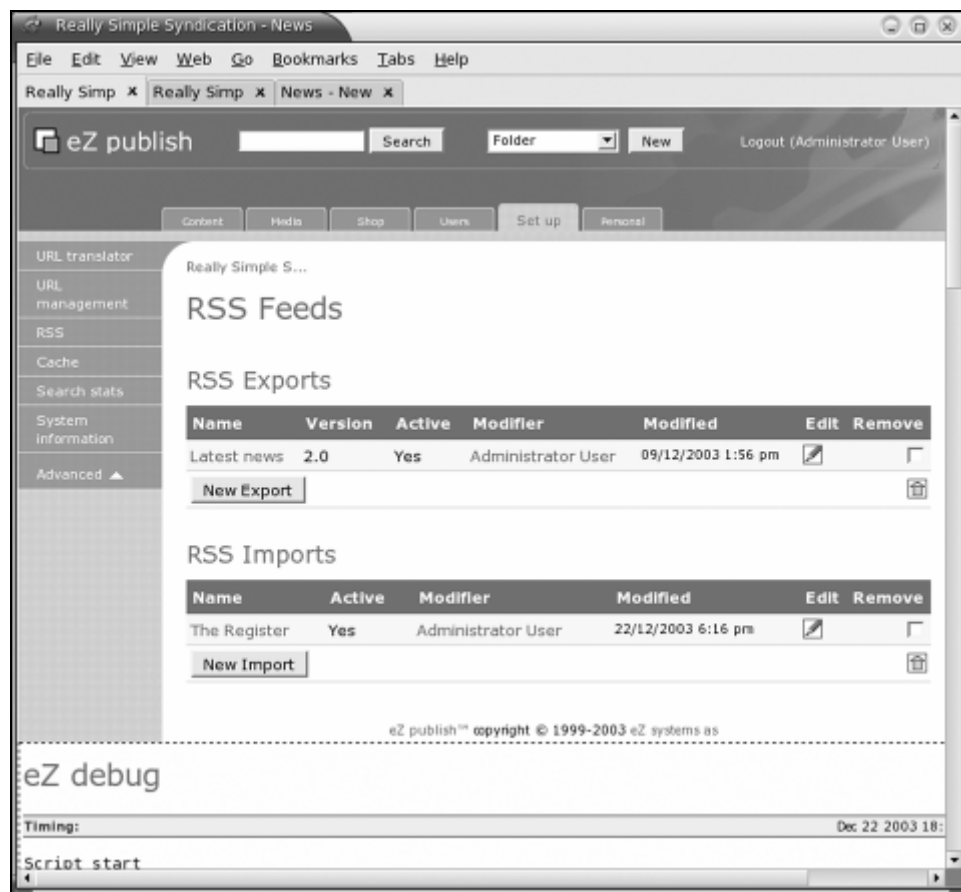
RSS (Really Simple Syndication) is a technology that enables the sharing of content from one server to another. You could say that this sounds like SOAP. This is true; there are similarities between the two technologies. However, RSS has been designed to fulfil a particular need whereas SOAP is more of a general-purpose protocol. At the time of writing, eZ publish v3.3 is in beta stage and has a full implementation of RSS 2.0. We will use this for our discussion.

The benefit of using an RSS import feed is that you can import the information directly into an article or other eZ publish class of your choice. No additional coding is required to implement this type of system using the eZ SOAP interface. This means that you can update the feed once at night via a cron job and that information will be available for your site visitors. This is a benefit when you cannot guarantee that the external RSS server will be available when visitors view your site.

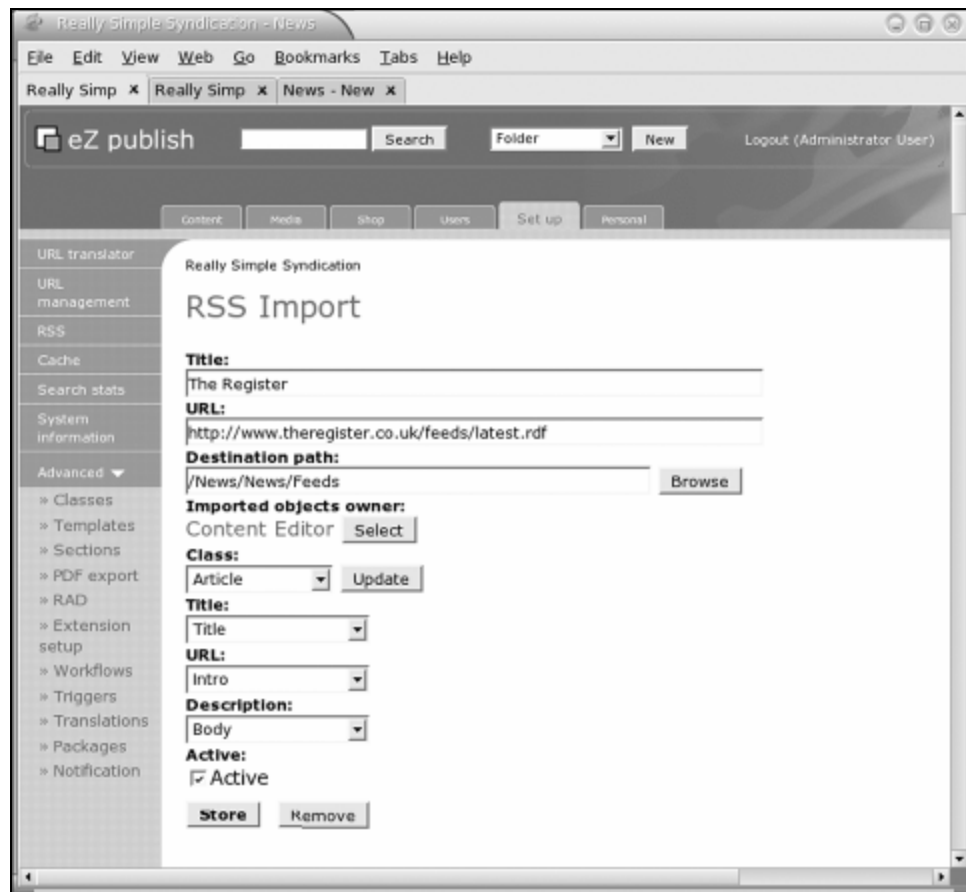
RSS feeds are used to give freshness to a site and help generate an element of "stickiness". This can be used in the traditional news format, where you get the RSS feeds from a news provider, such as *The Register*, allowing your site to benefit from the latest information and give it a fresh feel.



RSS is now simpler to set up via an RSS admin page in the Set up menu. This enables you to create RSS 1.0 and RSS 2.0 exports from your site by selecting the items you want to be shared. This means you can enable single or multiple RSS exports without writing a single line of code.



The same process is followed for RSS imports, where you can choose the files you want to import from the admin interface. This means you can import news from an RSS feed directly to your news section. In this example, we have chosen to take the RSS feed from *The Register* news feed from <http://www.theregister.co.uk/feeds/latest.rdf>.



A growing use of RSS is in the area of content syndication, where it is used to distribute information around an organization. One example of this would be when you have both an Intranet and Internet server and want to distribute information between the two. You could use the RSS protocol on the Intranet server to pull relevant articles off the Internet server. The Internet server can use the RSS export features that are configured to serve RSS feeds on a regular basis, thereby preventing your editors needing to enter the information on both servers. In some large organizations, this can represent a reasonable cost saving in time and effort for editors.

Data Interoperability

The import and export of information is a key component of any CMS. With its present feature set and extensibility, eZ publish is an attractive package for centralizing information from other systems and for providing it to others.

Version 3.3 of eZ publish has started to provide improved import/export functionality, allowing the system to interoperate with other systems in a much easier fashion. In addition to SOAP and RSS mentioned within this chapter, the new functionality includes class and template import/export and PDF export. Until now, it has been left to program developers to write additional code to move information around.

This section will discuss some of the issues regarding importing, and give an example of the code required to import information into an eZ publish object. The export of information is not covered explicitly because the same methods and eZ APIs are used for this. In addition, the previous sections on SOAP and RSS have covered examples of data export in detail.

Importing Information

The new package format, introduced in eZ publish 3.2 and improved upon in 3.3, allows for classes and templates to be imported and exported. Unfortunately, content objects are not yet dealt with, but it is relatively simple to add your own using the eZ publish API.

Importing information concerns setting values of attributes with the information you want to import. For example, if you want to import text from a document, you will need to import that text into the body attribute of an article. If you want to store category information, then several folder-content objects will be required, one for each category.

The following example will show how to create a simple folder-content object and then publish it.

Publishing a Folder Object

This example uses parts of the complete file `import.php` that is present in the download code for this chapter.

As with the administration interface, content objects are created from classes at the code level. The following code will retrieve a class instance given a class identifier as the parameter:

```
$contentClassID=1;
```

```
$class =& ezContentClass::fetch($contentClassID );
```

The class identifier value can be found in the administration interface; for example, a folder is always assigned the number 1 and an article is assigned the number 2. To create a folder, assign the value of 1 to the \$contentClassID variable.

```
$sectionID=1;
$userID=14;
```

```
$contentObject =& $class->instantiate($userID, $sectionID );
```

Then, we create a content object from the class by instantiating it. This builds an unpublished, empty content object. eZ attaches an object to a node and positions it within a tree structure, which can be coded as follows:

```
$nodeAssignment =
    & ezNodeAssignment::create(array(
        'contentobject_id' => $contentObject-
>attribute('id'),
        'contentobject_version' => $contentObject-
>attribute(
                                                    'current_version'
),
        'parent_node' => 2,
        'sort_field' => 2,
        'sort_order' => 0,
        'is_main' => 1 ) );
```

```
$nodeAssignment->store();
```

This code inserts the object into a node and places it within the root directory of the administration interface. The parent_node array key identifies the parent nodes, and therefore the object's location within the tree; in this example, node 2 is the root node.

With the content object created and assigned to a node, its version information should be updated. This allows eZ to display status information about this node.

```
$contentObjectVersion =& $contentObject->version($contentObject-
>attribute(
                                                    'current_version' )
);
$contentObjectVersion->setAttribute('status', EZ_VERSION_STATUS_DRAFT
);
$contentObjectVersion->store();
```

The current version of the content object is retrieved and the status is set as a draft. This status will be changed once the object is published.

The content object is now ready for information to be entered into its attributes. When the object is created through the class instantiation function call, each attribute is also created. At the moment these are empty:

```
$contentObjectAttributes =& $contentObjectVersion-
>contentObjectAttributes();
```

First, you need to retrieve a list of attributes from the current version of the object:

```
foreach (array_keys($contentObjectAttributes) as $key)
{
    $contentObjectAttribute =& $contentObjectAttributes[$key];
    $contentClassAttribute =& $contentObjectAttribute->
        contentClassAttribute();

    // Each attribute has an attribute called 'name' that identifies
    it.
    if ($contentClassAttribute->attribute("name") == "Name")
    {
        $contentObjectAttribute->setAttribute("data_text", "My new
        folder");
        $contentObjectAttribute->store();
    }
}
```

If you know the precise attribute you need, you can reference it directly as you would with any PHP array. However, for illustration purposes, this code loops through all attributes looking for the single attribute called `Name`. This attribute is a simple string, and the text `My new folder` is assigned to it. Other attributes will require additional code to assign information.

Note that it is at this point that the import has occurred. The text assigned could easily be the result of a function call to another routine, which has the information ready to be stored.

The final stage is to publish the object, which is done by the following function call:

```
$operationResult = ezOperationHandler::execute('content', 'publish',
    array('object_id' => $contentObject-
    >attribute('id' ),
    'version' => $contentObject->
    attribute('current_version' )
) );
```

Login Handlers

You already use a login handler for eZ publish, the default one. This enables you to gain access to your system and its content and features. However, there are times when you may need to integrate your system with an environment that already has a method of storing user details. If this is the case, you need to use a different login handler. This section will demonstrate the current options available to you.

Custom login handlers enable you to state that users are being managed externally. In addition, each user of a system will need to be defined within the eZ

publish system to enable access to the eZ publish system. This method allows you to manage the updates and access status of the user.

LDAP (Lightweight Directory Access Protocol)

LDAP has been supported in eZ publish since version 3.2 and incorporates the ability to customize the login handler such that it can use a LDAP server instead of the information stored with the eZ publish SQL database.

By default, the login handler is set to the default or standard handler in `site.ini`:

```
LoginHandler[]=standard
```

To configure your system to use the login information in the `ldap.ini` override file, you should change this to:

```
LoginHandler[]=LDAP
```

Sometimes, though, you may also want to allow the standard eZ publish logins to work for users such as the administrator. To enable this, list both login methods in the order you require them to be accessed in:

```
LoginHandler[]=standard
LoginHandler[]=LDAP
```

You will also need to override `ldap.ini`. You will need to contact the LDAP administrator to get the information required to ensure that your system connects to the LDAP system correctly. Once this information is set, the system will handle logins by using the `kernel/classes/datatypes/ezuser/ezldapuser.php` file. eZ publish integrates with LDAP by matching already existing user IDs in both your local eZ publish system and the remote LDAP server. This information is then updated from the LDAP server whenever the cron job `/cronjobs/ldapusermanage.php` is executed.

For the LDAP updates to work, initially you will need to add the user information to both eZ publish and LDAP systems. Once this is completed, the cron job will update the first name, last name, e-mail, user group, password, and the enabled status from the LDAP source.

This means the LDAP handler removes the need to manage users' details on the eZ publish system on a day-to-day basis, but you will still need to add new users to both systems.

You will need to open your firewall to allow the chosen LDAP communication port; by default this is port 389. PHP will need to have LDAP enabled for this method to work.

```
?ini charset="iso-8859-1"?
```

```
# ez publish configuration file for connection to LDAP server
#
[LDAPSettings]
# Set to true if use LDAP server
LDAPEnabled=true
# LDAP host
LDAPServer=
# Port nr for LDAP, default is 389
LDAPPort=389
# Specifies the base DN for the directory.
LDAPBaseDn=
# Could be sub, one, base.
LDAPSearchScope=sub
# Use the equal sign to replace "=" when specify LDAPBaseDn or
LDAPSearchFilters
LDAPEqualSign=--
# Add extra search requirement. Uncomment it if you don't need it.
# Example LDAPSearchFilters[]=objectClass--inetOrgPerson
LDAPSearchFilters[]
# LDAP attribute for login. Normally, uid
LDAPLoginAttribute=uid
# Could be id or name
LDAPUserGroupType=id
# Default place to store LDAP users. Could be content object id or
# group name for LDAP user group,
# depends on LDAPUserGroupType.
LDAPUserGroup=
# LDAP attribute type for user group. Could be name or id
LDAPUserGroupAttributeType=name
# LDAP attribute for user group. For example, employeetype. If
# specified, LDAP users
# will be saved under the same group as in LDAP server.
LDAPUserGroupAttribute=employeetype
# LDAP attribute for First name. Normally, givenname
LDAPFirstNameAttribute=givenname
# LDAP attribute for Last name. Normally, sn
LDAPLastNameAttribute=sn
# LDAP attribute for e-mail. Normally, mail
LDAPEmailAttribute=mail
```

Text File Login

With the introduction of the login handler, you can have other methods of authentication; one of the provided methods is to verify login by using a text file. This is essentially the same as LDAP, but it uses a file to store login information, and the `eztextfileuser.php` file as its login handler.

The layout for the text file handler can be found in `textfile.ini` in the settings directory.

Summary

It is clear that eZ publish is a complex system with myriad ways to extend it. We have discussed many of these situations and given examples of how to begin interaction with eZ publish on each of these levels. A very important part of this discussion was to identify and separate each extension area to enable you, the developer, to use the correct tools and extensions to build your application. In the next chapter, we will move on to practical examples of extending eZ publish using the methods introduced in this chapter.

6

Extension Development

In this chapter, we will look at some examples and concepts that put the contents of the previous chapter into real-world contexts as we tackle the general problem of extension development.

Two extensions are covered—one is a new e-commerce module for working with the WorldPay payment engine, which will enable you to implement payment solutions from your eZ publish site. The other is a category datatype—this helps to resolve the problem of category lists in articles or products.

Finally, we will discuss integrating external applications with eZ publish, and the problems and choices that you may have to deal with.

We begin with a brief summary of extension development practices in eZ publish.

Extension Development Practices

As with any development project, when creating an extension it is a good idea to have a clear idea of what you want to achieve, so get out the pen and pad and start writing!

Designing Your Extension

The development issues can be broken down into the following areas:

- Goals and targets of the extension
- Preparing to test your extension
- Timescales
- Anticipating the learning curve

Goals and Targets of the Extension

You need to define what you want to achieve by identifying key areas that are required for development. This allows you to understand the problem and build an outline of how you intend to resolve it.

Having a good understanding of the goals of a project can help prevent later problems, such as scope creep, which cost time and money in a project. This step lets you plan out the individual elements of the extension and understand general issues related to it.

Preparing to Test Your Extension

Take the listed goals and from them build a brief plan of how each element is to be tested. At the end of the project, this will help you confirm that you have created exactly what you needed.

Timescales

Identifying time limits and time issues early on paper focuses the mind and helps you figure out the 'nice to have' features as opposed to the mandatory elements of your extension.

Key elements are client deadlines, the time available for development, and eZ publish release schedules. These schedules, if late, could force you to deliver a project using beta software, so always try to keep a month's buffer after the release date if you plan to roll out an extension, as most first releases contain at least some bugs.

Bear in mind that you will be writing an extension with the eZ publish libraries, so you will need to add a little extra time in the first couple of projects so you can learn the libraries and the various integration methods.

Anticipate the Learning Curve

If you cannot resolve all the problems yourself, you may need help from others. Support from the forums can take a while if you are not paying for it. Even in the basic paid support package, issues normally take three days to solve, so plan carefully based on your needs.

Bear in mind that around major eZ publish release times, free forum support can be affected because eZ staff and developers are working on the latest release.

Software Requirements

If you use a feature in your extension that is only available in the latest version of eZ publish, but the project uses a previous version, you will probably experience a problem.

Remember that not all hosting environments support the latest version of PHP and MySQL; always ensure that your development environment is the same as your deployment platform.

It is always a good idea to test all new features on your deployment platform early, as this prevents problems during testing.

Development Tools

Various tools can be used in the development of an extension. **CVS** (Code Versioning System) and **SVN** (Software Version Number) are invaluable in tracking your code changes and have been incorporated into many integrated development environments. Some recommended open source IDEs that work well with eZ publish are **Eclipse** (www.eclipse.org) and **Quanta Plus** (<http://quanta.sourceforge.net>).

If these two IDEs are not to your liking, it is worthwhile spending time looking for a tool you feel comfortable with that works with HTML, PHP, SQL, and eZ publish template files, as you will be regularly changing all these file types.

Sharing with the Community

eZ publish has a dual license that enables you to share or not share depending on whether you pay eZ publish a development fee.

The eZ publish dual license gives you two options. If you are not paying eZ systems for a developer license, you need to make your code public. See a copy of this on <http://www.gnu.org/copyleft/gpl.html>. This means that any extensions you write that use eZ publish libraries or functions must themselves become GPL. This involves adding the GPL license information to each of the programs you wish to share. There are several ways for sharing your extension:

- eZ publish contributions
- Public SVN
- SourceForge

The recommended choice is to contribute via the eZ publish site and add your contribution (<http://ez.no/community/contributions>). We suggest you upload your contribution here first. Once it is added, you can let people know by posting a message to the developer forum saying that you have written an extension and providing a brief description. Although this is not strictly necessary, it does start a thread for people to discuss the extension. You will generally get good feedback from the community and someone could help develop your contribution further.

Public SVN is a new location set up to enable community projects to be hosted centrally in a change-controlled format (<http://pubsvn.ez.no/community-development.html>). At the time of writing, this is a very new service, so try to visit the website.

SourceForge has many open source projects available for download and contribution and is a good choice if you want to share and actively develop your extension without hosting it yourself. However, move to SourceForge only after reviewing the first two options, so that eZ publish developers can have a central source for all extensions.

Upgrading eZ publish

Future major releases of eZ publish could well take advantage of the new features available in forthcoming versions of PHP and MySQL, so this is something worth bearing in mind.

It is worth making a note of the major requirements of your extension so that you can quickly check for any functionality changes with any new release of eZ publish.

When these updates do occur, there will be guidelines from eZ systems and from the community to help developers.

Documentation

If you have ever gone back to review a project or extension after a period and not been able to decipher how it works, or it has taken you a long time to figure out where a problem is, then you are a prime candidate for documentation. This subject usually evokes the greatest procrastination from developers! Unfortunately, documentation is essential and you're going to have to do it. Some key areas for documentation include:

- Overview of the goals of the extension
- Requirements
- Installation instructions
- eZ publish version
- PHP version
- Database requirements and setup

A documentation tool has been integrated into eZ publish to help you document your code. It is called **Doxygen** (<http://doxygen.org>). This is one of the many documenting tools that are freely available and helps take the tedium out of documenting.

Creating the WorldPay Extension

In the previous chapter, we discussed all the elements of an extension. In this section we will take these ideas and concepts and apply them to something virtually every e-commerce site needs—a payment engine.

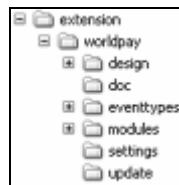
This example uses the WorldPay Direct account (<http://www.worldpay.co.uk/sme/>) and will communicate with the WorldPay service using the WorldPay Junior integration service (<http://support.worldpay.com/integrations/jnr/index.html>). Most

payment services are similar so this forms a good base for all payment engines. Once you have looked through the code you will be able to see where it can be modified to work with other payment services. The WorldPay module requires that we use both events and modules together to get the payment engine to work, as we need to integrate with the checkout workflow from both a user and a remote-call perspective.

Creating the Environment

If you are recreating this example on your own machine, it is assumed that you have eZ publish up and running on it. This step involves installing the latest eZ publish system, running the setup, and following the wizard so you can see the site and administration screens correctly. This is covered in Chapter 2 and will not be duplicated here.

Think of an extension as an application that uses the eZ publish environment, and for it to be effective, you should have all the files you need within your project directory. This will ensure extension mobility—you will be able to take the written code and move it to another project easily. If you take the directory structure as the starting point, your directory will look like this:



The following table contains an example of the content required for each directory:

Directory	Contents of the Directory
design	All stylesheets, JavaScript, images, and eZ publish templates.
doc	All the documentation required for someone to install and use the extension.
eventtypes	New custom events that you are creating.
modules	New modules that the project uses.
settings	Any new ini settings required by your project.
update	Any updates to eZ publish that are required. This includes database and code patches.

In this extension, we are not using multiple languages, new datatypes, or actions, so we do not need to include their directories.

The main eZ publish environment needs to know that an extension is available and should be made available to the site. For this, change the `site.ini.append.php` file. Add the following settings to activate the extension:

File: `settings/override/site.ini.append.php`

```
[ExtensionSettings]
ExtensionDirectory=extension
ActiveExtensions[]=worldpay
```

This lets eZ publish know that you have an extension called `worldpay` that it needs to know about. If you have ever used the `ezdhtml` extension for the online editor, you will know that these settings are needed as you need to add similar commands for eZ publish to see the editor.

eZ publish will now look into this directory for any overrides that the extension might need.

Creating Workflow Events and Triggers

In the following example for a payment engine, we want the action to take place before the checkout event has occurred and the items are all in the baskets ready for payment. Before we can select a dropdown from the workflow list, a workflow needs to be created.

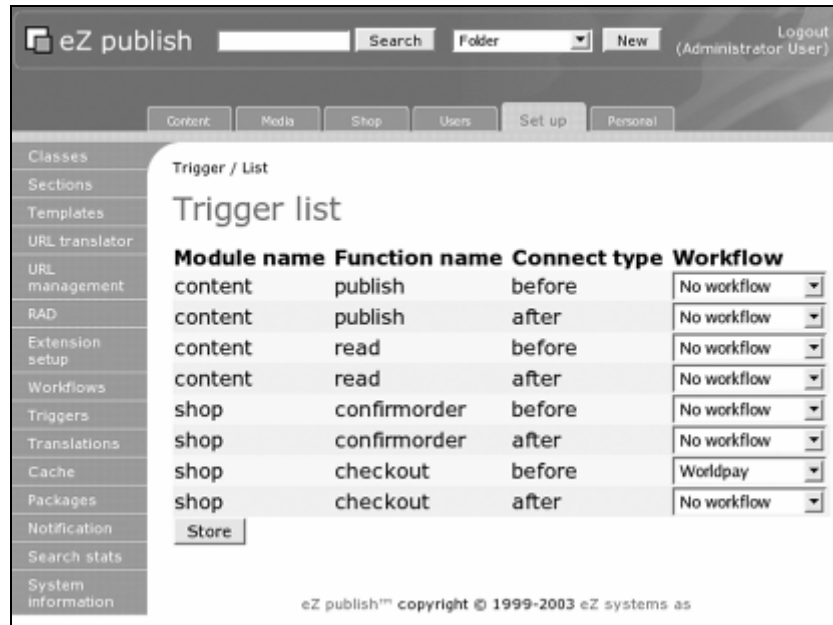
To create the Worldpay workflow:

1. Click on Workflows in the set-up menu. This will show you a list of existing workflows. To this, add a new group called `worldpay`.
2. Once this group has been created, click on the Worldpay group link to view the available workflows. As this is a new group, there will be no workflows, so you need to create one by selecting the New workflow button.
3. To name the workflow, add `worldpay` to the Name field.
4. Select Event/Worldpay from the drop-down event list at the bottom of the page and click the New button.

If you do not see the Worldpay item in the list, verify that you have declared `worldpay` as an extension in the `site.ini` files.

5. You do not need to enter a description for the event, so go ahead and click Store to save the workflow.

Now that the workflow event has been created, you need to come back to this screen by selecting Setup | Triggers and select Worldpay next to shop | checkout | before.



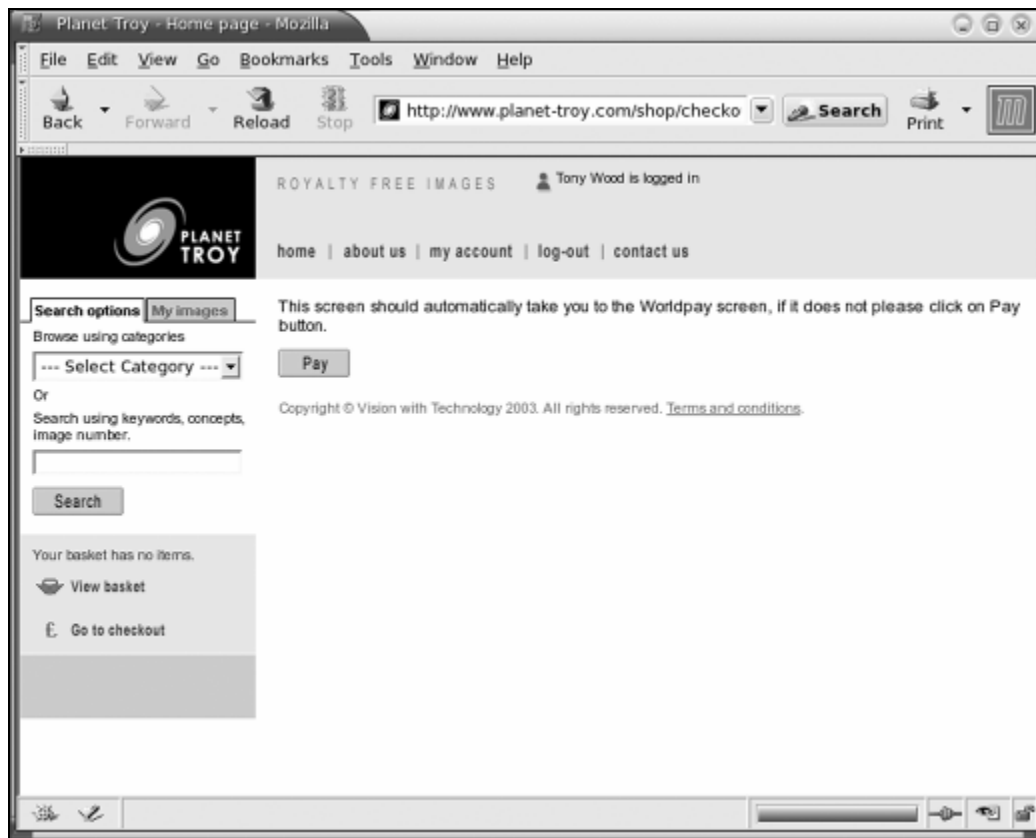
6. Create the eventtype extension environment. As this workflow will be part of a larger project, it will need to live somewhere. Create the extension directory (if you don't already have one) in your project directory. Then create a directory `worldpay` below it:



7. Create the `worldpay` eventtype. The eventtypes directory contains the program that gets executed when a trigger is activated. The name of the file follows the format of `nametype.php`.

Overview

The `worldpay` eventtype needs to take the current checkout information and create the form screen that is required for input to the WorldPay system. This involves taking the price and billing information from the basket and sending it to WorldPay as an HTML form post.



PHP Implementation

Take a couple of moments to review the `worldpaytype.php` source file. As you can see, there are various includes and definitions at the top, but the most important elements you need have been stripped out and shown here:

```
<?php
define( "EZ_WORKFLOW_TYPE_WORLDPAY_ID", "worldpay" );
class WorldpayType extends ezWorkflowEventType
{
    /*!
    Constructor
    */
    function WorldpayType()
    {
        $this->ezWorkflowEventType(EZ_WORKFLOW_TYPE_WORLDPAY_ID,
                                   "worldpay" );
    }
    function execute( &$process, &$event )
    {
        .....
    }
}
```

```
eZWorkflowEventType::registerType(EZ_WORKFLOW_TYPE_WORLDPAY_ID,
                                "worldpaytype" );
?>
```

The WorldPay example is slightly more complicated than a normal event as it is called two separate times—once by the customer when they click the checkout button and again by the WorldPay servers in their callback routine that tells the eZ publish system that the credit card has been accepted and the transaction is a success. As an extra check the amount paid and other variables sent to WorldPay are also returned so the WorldPay extension can confirm these details as well.

The new workflow routine does the following:

- Retrieves order information from the eZ basket or the HTTP POST.
- Uses the information retrieved from the previous bullet to interrogate the eZ database to see if this order is new or old.
- If this requires a new order, creates an HTML form pre-filled with information required by WorldPay for payment and processes workflow for this event.
- If this is an existing order, checks whether the order has been approved by the callback routine. If yes, the workflow is finalized and the order status moved from 'in progress' to 'complete'.

In this example, there is an extra file called `worldpaydb.php` that handles all communications with the persistent datastore, in this case a MySQL table. This is to ensure that this module is consistent with the eZ publish object-orientated method of database communication. It also ensures that all data manipulation functions are stored in one place.

You could just as well use a PostgreSQL database, but this example has only been tested on MySQL 4.0 and 4.1 (alpha).

ini Settings

The WorldPay example has the standard set of overrides required for an extension to work as discussed in previous chapters, but it also contains its own extension-specific `ini` file (`worldpay.ini.append.php`) that stores information peculiar to this extension.

Problems

At the time of writing, there is a problem: the basket is cleared when the first checkout routine is run. This is an issue if a customer does not complete the payment stage and wants to return to their basket to continue shopping.

This problem occurs because the system uses the current session key for the operation memento data key; this means that the basket will not be cleared upon checkout if the operation is not continued from the browser session that added the items to the basket. This is the case as the order is continued and completed from the WorldPay servers.

A future fix for this would be to send the current session to the WorldPay server and then get it back in the callback. When you get it back, you set it as the current session using `eZHTTPTool::getSessionKey()` and run the operation as normal.

For now, the WorldPay routine uses an ini setting in `site.ini` that is required to resolve this issue.

If you do not add this setting, then there is the problem with order clearing that enables customers to add items to the basket after purchase, which will then appear in the order as purchased. Not a very nice situation for the merchant!

```
[ShopSettings]
# Whether to clear the basket on checkout or not
# if disabled the basket will be cleared after the checkout is
# complete
ClearBasketOnCheckout=enabled
```

WorldPay Module

Although modules are not necessary for some smaller projects, they are required for doing anything extensive. Modules are powerful, and like eventtypes, they enable you to interact with every part of eZ publish from templates to roles.

In this example, we will take the knowledge gained from the previous chapter and create the **callback** feature for the WorldPay module. Callback is a well-used mechanism that helps you ensure that your payment provider has approved the purchase and allows you to check exactly what has been purchased. This is an added feature that helps prevent fraud. Some payment mechanisms do not provide this feature and it is a simple matter for a fraudulent user to change the amount required and get the goods at a reduced price.

To enable callback within the WorldPay system, modify your WorldPay settings in the CMS (WorldPay's Customer Management System). Select the configuration options page and enable **Callback enabled** and **Use callback response** and add the URL to your site into the **Callback URL**:
`www.example.com/worldpay/callback`.

Both PayPal and WorldPay use this mechanism, but as this is a WorldPay module, we will discuss how to enable this with WorldPay.

Creating the Module Extension Environment

As this module will be part of a larger project, it will need to live somewhere. So create the extension directory (if you don't already have one) in your project directory. Then create a directory called `worldpay` below it.



Creating the Module

This extension requires a page for WorldPay to call back to when the payment transaction is finished. So, you need to define a page called `callback` so the WorldPay servers can post information to `www.myserver.com/worldpay/callback`.

To create the callback page, the following files are required:

- Module definition file (`module.php`): This file informs the eZ publish system what scripts are available in this module.

```

<?php
    $Module = array( "name" => "Worldpay",
                    "variable_params" => true,
                    "function" => array("script" =>
                                        "callback.php") );
    $ViewList = array();
?>

```

- Function collection file (`worldpayfunctioncollection.php`): This is a dummy file in the example, but normally it contains any functions that eZ publish needs to know about.

```

<?php
    // debug
    include_once( "lib/ezutils/classes/ezdebug.php" );
    // Order list
    include_once( "kernel/classes/ezorder.php" ); are these
                                                    required?
    include_once( "kernel/classes/ezproductcollectionitem.php" );
    class WorldpayFunctionCollection
    {
        function WorldpayFunctionCollection()
        {
        }
    }
?>

```

- Your module file (`callback.php`): This file contains the code that checks the post from WorldPay and sends an HTML response, which WorldPay then displays. This enables the end user to confirm that they have purchased item(s) or tells them that an error has occurred and they need to speak to customer service.

The `callback.php` file and workflow routine use additions to the standard user class, so please read `manual.txt` in the extension for the latest information before installation.

Throughout the module you will see `eZDebug`. This displays your debug messages if you have `DebugOutput=enabled` in your `site.ini`. It is worth using instead of `printf()` or other functions, as it will format the output and also display any variable you throw at it.

As a general tip, to see what posted variables values are being sent to you, in this case by the payment engine, you may find the following snippet useful:

```
#check calling parameters
$y="";
foreach ( $_POST as $x )
{
    $y=$y.", ".$x;
}
$tpl->setVariable( "Params", $y );
```

You will need to pass this variable to `callback.tpl` so the result will be visible to you.

If you look at the source, you will see that the program starts by getting the variables passed to it. This is handled by the `eZHTTPTool` `eZ` library routine, which is used to query POST and GET variables. As shown in the following example, `http` is used to store the HTTP variables, which are then sent to the functions for processing:

```
$ini =& eZINI::instance( "worldpay.ini" );
$http =& eZHTTPTool::instance();
$tpl =& templateInit();
$status=VWT_PAYMENT_FAILED;
if ( !isWorldPayCall( $ini, $http ) )
```

These variables are then passed to the check functions used for payment validation.

Note that the payment approved status (`$status`) is set to `false` at the beginning of the `callback.php` script and is only changed when all the checks have been completed and the payment is shown to be acceptable.

We'll now look at some of the key functions in the extension that handle most of the payment functionality:

- `isWorldPayCall()`

- `isAmountOK()`
- `isPaymentOK()`

WorldPay has a range of servers that send responses, so a check needs to be made to prove that the IP address comes from these servers. The `IsWorldpayIPOK` function called from `isWorldPayCall()` uses the settings in the `worldpay.ini.append.php` file to figure out the correct settings. You will need to check these with WorldPay and don't forget to add your own development IP range here, else your routine will fail every time. If you have debugging enabled, you will spot this problem early on. The other check handled here is verifying that the installation ID is correct—this ensures that you are working with the correct WorldPay installation. This is useful as the Junior service consists of two installations and it is best not to get them mixed up.

The `isAmountOK()` function verifies that the payment approval matches the amount you sent to the payment engine. This prevents people from falsifying the call from WorldPay by changing the amount to a lower figure without your approval.

The problem with checking the price is that most of the time you will need to handle more than one currency. This routine needs to:

- Check the exchange rate for the currency
- Convert it back to your site's pricing currency
- Ensure that the difference is within a defined limit

If the price difference is greater than the `PriceDifference` set in the `worldpay.ini.append.php` file, the order will not be approved, but the money is still taken from the credit card. In such a situation, it is likely that you will get a customer service call from the customer, and you will need to resolve the situation manually.

This differential amount can be set by you in the `worldpay.ini.append.php` ini file as follows:

```
#Price difference +/- allows when currency is converted
PriceDifference=1.00
```

The exchange rates used for this routine are available from WorldPay and if you have access to a cron job, they can be retrieved and stored in an eZ publish directory using the following routine:

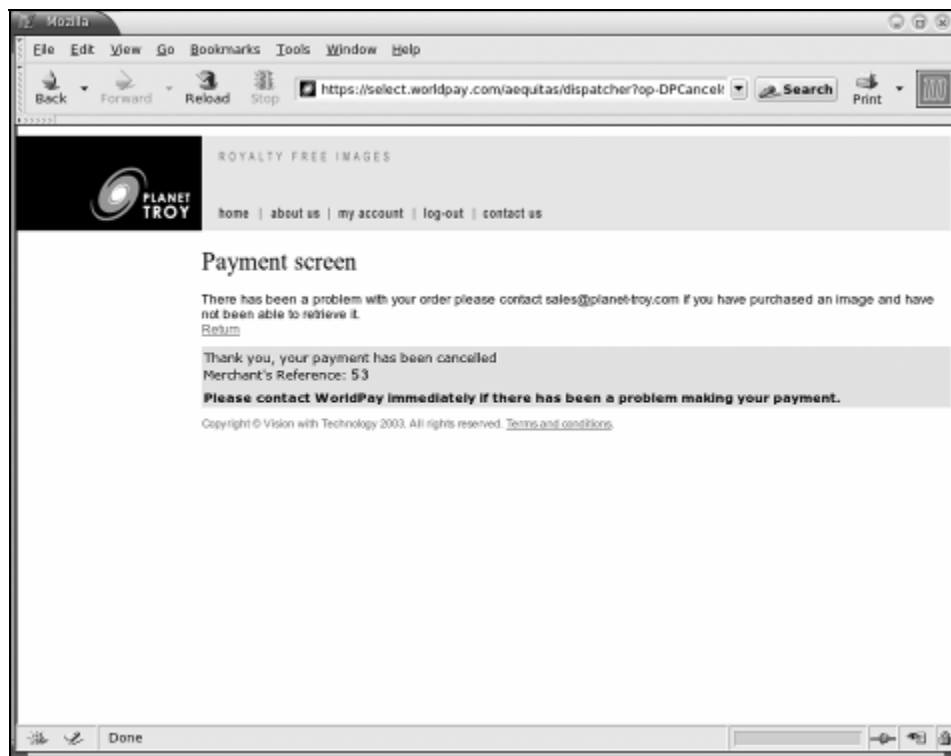
```
>wget 'https://select.worldpay.com/wcc/info?op=rates&instId=99999'
-O /path/to/worldpay/module/xrate.ini.append.php
```

The `isPaymentOK()` function goes through the final checks in the module. It confirms that the information stored earlier by the workflow eventtype matches the information sent to you by the WorldPay servers. If the information is correct, the payment flag is set to approved.

If the payment is deemed correct, the workflow needs to be marked as completed. This is achieved by the following command that calls the workflow giving it the order ID from the order. This will complete and clear the workflow from the system and create the completed order information that is shown in the order list within the admin section of the system:

```
$operationResult = eZOperationHandler::execute( 'shop', 'checkout',
    array( 'order_id' => $order_id ) );
```

If the order is not approved or deemed incorrect, the module will return an error message relating to the type of error. In this example an error is created after a customer hits Cancel order in the WorldPay screens:



When the workflow runs, it marks the order as complete and fires off the order e-mails you love if you run an online business! Finally, you need to let the customer know the outcome of their transaction. This step is completed by sending a blank template `wp_pagelayout.tpl` filled with the outcome and necessary details. WorldPay has a surrounding menu structure so we need a new template file to ensure that just the response is sent to WorldPay and not the menu and navigation:

```
$Result['pagelayout'] = 'wp_pagelayout.tpl';
```

Reviewing the ini Settings

It is worth taking a moment to review the ini settings in the extension:

File	Description
<code>worldpay.ini.append.php</code>	Stores the password, installation settings, price difference, and WorldPay server ranges
<code>content.ini.append.php</code>	Contains the available extensions and datatypes
<code>design.ini.append</code>	Specifies the design extensions used
<code>layout.ini.append</code>	Specifies the page layout to be used in this extension
<code>module.ini.append</code>	Defines modules available in this extension
<code>workflow.ini.append</code>	Lists the eventtypes used in this project

User Settings

The extension code uses a modified user class. To get the extension to work, you need to make a couple of changes to the user class. The main change is using a `matrix` datatype for storing the address, because you can never know how many lines an address will be, especially if you consider country differences.

The `worldpaytype.php` file requires the user class to contain the correct attributes. These are, in order:

Name	Datatype
First name	Text line
Last name	Text line
User account	User account
Company name	Text line
Address	Matrix
Telephone Number	Text line
Country	Country or Text Line

Once the class is changed, you need to ensure that the definition in `worldpaytype.php` is set up. Here is an example:

File: eventtypes/event/worldpay/worldpay.php

```
define( "VWT_FIRST_NAME", 0 );
define( "VWT_LAST_NAME", 1 );
define( "VWT_USER", 2 );
define( "VWT_COMPANY", 3 );
define( "VWT_ADDRESS", 4 );
define( "VWT_TEL", 5 );
define( "VWT_COUNTRY", 6 );
```

Note that Country is an unpublished datatype but is based on the `ezcountry` datatype in http://ez.no/community/contributions/datatypes/datatype_ezcountry. If you do not want to use the Country datatype, you can use a text line for the country instead.

If you do not want to use the main user class and instead create a new class for shop users, read the following online documentation:

http://ez.no/developer/ez_publish_3/documentation/incoming/tutorial_using_userregister.

This document explains some of the issues surrounding users and registration.

Permissions

You need to give shop users access to the WorldPay extension via the roles administration screen (<http://www.example.com/role/list>). This ensures that they have access to the templates and PHP.

You also need to ensure that the WorldPay servers have permissions for callback. For this you need to allow anonymous users access to the WorldPay extension.

Callback Testing

The one area that can be difficult to test is the callback element of the extension. This is because you need a live URL for the WorldPay server to send back its response. For this reason the `wpcheck.html` file is included to recreate the WorldPay server callback HTTP POST.

You will need to set the URL to your test server and the Installation ID in the file. Then for each test you carry out you will need to get the following information from your checkout screen. Just remove `setTimeout( 'document.worldpayForm.submit()', 0 );` from your `event_worldpay.tpl` and it should be a simple matter to cut and paste the variable information into the `wpcheck.html` file to make it ready for use!

File: doc/wpcheck.html

```
<input type=hidden name="M_email" value="nosпам@visionwt.com">
<input type=hidden name="M_PHPSESSID"
value="555rrc54442a048b6152444d38d8b">
```

```
<input type=hidden name="M_USERID" value="147">
<input type=hidden name="M_ORDERCREATED" value="1067446530">
```

All you need to do now is run `wpcheck.html` from your browser once the checkout workflow has been enabled and you have a quick and safe way of testing callback without the risk of developing on a live URL.

Creating the Category Datatype

The need for a category datatype arose during the development of an e-commerce site using the WorldPay module. In this project, each product belongs to a category that is used for search results. The use of other datatypes such as enumerations and text lines was considered but they have inherent problems. Existing datatypes require that category name changes be made to the class. After publishing the change to the class, eZ publish does not automatically update the affected objects with the new information—this must be performed manually. With the e-commerce site, the number of products/objects was (and still is) considerable and would have incurred a large amount of non-budgeted work each time a category was altered.

The category datatype prevents this situation because the update of the category is moved from the class to the object.

Category Datatype Design

This datatype addresses the main issue of allowing objects to add and remove categories as they wish. The design of this datatype includes the use of a new SQL table and new PHP classes that utilize eZ functions to communicate with the new table.

Setting Up the Extension Environment

The category datatype is actually a separate extension from WorldPay, but from the systems point of view both extensions are active.

First we inform eZ publish that the extension is active, from the main configuration file:

```
File: settings/site.ini
[ExtensionSettings]
ActiveExtensions[]=category
```

Within the extension, the system must be told about the new datatype and the design directory it uses. For the category include:

```
File: extension/category/settings/content.ini.append
[DataTypeSettings]
ExtensionDirectories[]=category
AvailableDataTypes[]=category
```

For the design directory, include:

File: `extension/category/settings/design.ini.append`

```
[ExtensionSettings]
DesignExtensions[]=category
```

The Category Database Table

This new database table will associate categories using the attribute ID, which belongs to the object that the category datatype is part of. A content object attribute ID can be used to find the content object itself, and within the context of the e-commerce system, objects using this datatype are classed as **products**.

You need to create the table in the MySQL file using the provided SQL create table script:

File: `extension/category/update/mysql/category.sql`

```
CREATE TABLE category_information
(
    id integer NOT NULL auto_increment,
    category varchar(255) NOT NULL,
    objectattribute_id integer NOT NULL default '0',
    PRIMARY KEY (id)
);
```

In this table the column `id` is set as the primary key, ensuring that each new row will be unique. The `category` column is a string with a maximum length of 255 characters. The final column, `objectattribute_id`, is used to record the identifier of the attribute that corresponds with the datatype created, and the object it resides within.

When a category is added to this table, it exists as a row within the database. The following table shows some sample content for this database table:

id	Category	objectattribute_id
53	Nature	159
54	Work	159
55	Miscellaneous	159
56	Miscellaneous	165

In this table there are three categories associated with attribute 159. Attribute 165 contains only one category, but it is the same as the previous attribute. This structure allows each product to update the categories associated with it.

Please refer to your database documentation to learn how to add this table to your eZ publish system.

Database Communication

The eZ publish system defines a database object as a persistent object. Utility classes are provided that perform the mechanics of communicating with the database at an abstract level; this allows the user to ignore the type of database being used. By extending the parent `eZPersistentObject` class it is possible to define variables that correspond to columns in the category table on a one-to-one basis.

The category PHP file created for this purpose is very long and, for the purpose of this discussion, only the function names will be shown. Please refer to the code download to see the complete code.

File: extension/category/lib/category_db.php

```
class PersistentCategory extends eZPersistentObject
{
    function PersistentCategory( $row )
    function &definition()
    function &remove( $id )
    function &fetch( $id, $asObject = true )
    function &fetchByCategory( $category, $asObject = true )
    function &fetchByCategoryAndAttribute( $category,
                                           $objectattribute_id,
                                           $asObject = true )
    function &fetchByAttribute( $objectattribute_id, $asObject = true )
    function &fetchAllCategories( $asObject = true )
    function &create ( $category, $objectAttributeID )
    function hasAttribute( $name )
    function &attribute( $name )
    var $ID;
    var $Category;
    var $ObjectAttributeID;
}
```

The `definition()` function is the key to the database communication. Each persistent object implementation must implement this with a definition that matches variables to columns. With this in place, the `fetch()` functions can query the database. For example, the `fetchByCategoryAndAttribute()` function will return a database object that matches the category and attribute ID values that are passed in as parameters.

The functions `create()` and `remove()` respectively create and delete a database row. The `attribute()` function provides template access to the persistent object, which will be discussed later. The variables `$ID`, `$Category`, and `$ObjectAttributeID` correspond to the columns of the database tables.

Category Discussion

In addition to the regular `Category` type class for this datatype, there is an additional PHP class named `Category` that is used to fetch and store category information from the database. There will be no use of class attributes for this datatype, but if you feel a default value is needed, then by all means add your own.

Normally the fetching and storing of user input is performed by accessing the content object attribute or the content class attribute. Instead, here the content of the attribute is set to the value of the instantiated category.php object. For example:

File: extension/category/datatypes/category/categorytype.php

```
function fetchObjectAttributeHTTPInput( &$http, $base,
                                         &$contentObjectAttribute )
{
    if ( $http->hasPostVariable( $base . "_Category_data_"
                                . $contentObjectAttribute->attribute( "id" ) ) )
    {
        $data =& $http->postVariable( $base . "_Category_data_"
                                    . $contentObjectAttribute->attribute( "id" ) );
        $category = new Category();
        // parse the input data into the category array
        $categories = explode(",", $data);
        // add categories to the category class for later storage
        $category->initialiseCategory( $categories );

        // inform the content object of this content.
        $contentObjectAttribute->setContent( $category );
        return true;
    }
    return false;
}
```

The information passed in from the template is first processed through the PHP `explode()` function, so that multiple categories can be entered by means of a single comma-delimited string. Thus, a string such as "nature, work, miscellaneous" will be split into three different categories.

The Category class is initialized with this array and the content of the attribute is set with this class. When the system invokes the `storeObjectAttribute()` function, the categorytype will try to store the new information through the use of the new class:

File: extension/category/datatypes/category/categorytype.php

```
function storeObjectAttribute( &$contentObjectAttribute )
{
    $category =& $contentObjectAttribute->content();
    if ( is_object( $category ) )
    {
        $category->store( $contentObjectAttribute );
    }
}
```

The category `store()` function performs a few simple checks before entering the information into the database:

- If the category already exists for the attribute, there is no need to re-enter it.
- If the category is not present, it is entered as expected.
- If the database contains categories that are not part of the input list, they are removed.

File: extension/category/datatypes/category/category.php

```
function store( &$attribute )
{
    // Assign the attribute id
    $attributeId = $attribute->attribute( 'id' );
    // Get present categories for the attribute
    $existingCategories =&
        PersistentCategory::fetchByAttribute($attributeId);
    // Find out which categories to remove, if any
    foreach ( $existingCategories as $existingCategory )
    {
        // Check whether the current category is part of the new list
        if ( !in_array($existingCategory->attribute('category'),
            $this->CategoryArray) )
        {
            PersistentCategory::remove($existingCategory->
                attribute('id'));
        }
    }
    // Find which categories to add.
    foreach ( $this->CategoryArray as $newCategory )
    {
        $newCategoryObject =&
            PersistentCategory::fetchByCategoryAndAttribute($newCategory,
                $attributeId);
        // Test if the present category exists. If not add it as new.
        if ( !is_object($newCategoryObject) )
        {
            $createNewCategoryObject =&
                PersistentCategory::create($newCategory,
                    $attributeId);
            $createNewCategoryObject->store();
        }
    }
}
```

The use of the Category class for the content for the attribute also allows the template to access it. This class implements the following API for attributes:

File: extension/category/datatypes/category/category.php

```
function hasAttribute( $name )
function &attribute( $name )
```

This associates attributes that a user has access to within a template to functions that return the requested information. For the category class, the `category_string` attribute returns a string listing the categories the attribute belongs to in a comma-delimited fashion. It does this by querying the database for categories belonging to a particular attribute ID using the function `fetchByAttribute()`. Once it has this list, it uses the PHP `implode()` function to concatenate the list into a single, comma-delimited string:

File: extension/category/datatypes/category/category.php

```
function &categoriesString()
{
    return implode( ' ', $this->CategoryArray );
}
```

Category Templates

Two templates are required for this datatype. One is used to edit the datatype within a content object:

File: `extension/category/design/standard/templates/content/datatype/edit/category.tpl`

```
{let
name=concat("ContentObjectAttribute_Category_data_", $attribute.id)
value=$attribute.content.category_string}


```

The other is used to display the categories as a simple string:

File: `extension/category/design/standard/templates/content/datatype/view/category.tpl`

```
{ $attribute.content.category_string|wash(xhtml) }
```

The Category Datatype in Action

Now that we've seen the construction of the Category datatype, let's have a quick look at it in action before we move on. The following screenshots show how you can edit the class and object, and then view the object.

Editing the Class

6. Category (Category) (id:155)		
Name	Identifier	☐ Required ☒ Searchable ☐ Information collector ☐ Disable translation
Category	category	  

Editing the Object

Category
examples, category, few, a

Viewing the Object

Root folder / Article with category
Article with category [Article]
Category test
examples, category, few, a
Edit Preview Remove Bookmark Keep me updated

Integrating Existing Code with eZ publish

One of the questions faced by developers is "Do I create a new application or hack an existing one?" There are pros and cons for both, but since you're reading this book it is assumed you prefer to use a solid base and then customize it, rather than working on an entire system from scratch. This philosophy can continue into your extension writing as you can use the skills gained in previous chapters to build links to other applications. If you plan to do this, read on.

Bear in mind that when you link to another application, you immediately double the complexity of your application and its support requirement. It is a lot of work to maintain and update an eZ publish site with demanding customers; if you also have to update and maintain other applications and any middleware between them, it could prove to be too much!

When choosing to bridge to another application, ensure that it is not faster to write routines or customize some of the native functionality in eZ publish than to link to a best-of-breed package for features such as forums. This may be the case when choosing to integrate an existing best-of-breed forum package like phpBB with eZ publish.

With eZ publish and PHP you can pretty much integrate with anything—after all eZ publish is a development framework and as such perfectly extensible. As you will have read in previous chapters, you can make individual datatypes, eventtypes, and modules, and the WorldPay extension is an example of integrating a supplier's existing code with eZ publish. This means you are well equipped to handle all that is thrown at you!

Making a Bridge to External Applications

External applications could include operating systems, PHP applications, compiled applications—pretty much anything! If you can connect to the application from your system, for example using TCP, UDP, Unix sockets, or named pipes, then you can bridge to it using eZ publish. Some common examples for integration are:

- External systems via SOAP
- RSS feeds
- Databases: MySQL, dBase, Oracle
- Accounts systems
- Application service providers
- Legacy systems
- E-mail
- Open instant messaging systems, such as Jabber
- .NET integration

These are just a few examples of the types of systems you can integrate with. The biggest problem you will face is that some systems do *not* want you to integrate with them. This is because they have agreements with other proprietary systems or partners that want money before you can integrate. They may require you buy their 'integration add-on' before you can connect to their system. It is worth checking the costs of any add-ons you may need before you start out, as you may find them prohibitively expensive. The good news is that most of the add-ons for PHP are free so again it is worth staying with open source solutions if you can.

Let's take a couple of the items from the list and explore the possibilities.

Strategies

You will come across several strategies when working with external systems—the good news is that they are not restricted to eZ publish, and relate to all systems that need to get information from an external source. If you have an application on your desktop that goes to the Internet several times a day to collect your e-mail via POP3 you are using one.

With strategies, you have two basic choices:

- Import data to a local database
- Connect to external data for every request

The first choice is to take the external data and import it into your eZ publish database environment. This is a good choice if the link to the external site is unreliable or slow, or you are taking information from an external source that does not allow connection from the Internet. One example of this could be a site that uses information from a legacy system such as an ERP system. This would mean that you get the prices of products updated each day via a text file upload. You would have a routine that would run once the file is uploaded and perform the following functions:

- Check that the file has been uploaded
- Check the validity of the data
- Backup existing data prior to import
- Import the data into a special price table or update the product prices directly
- Update logs with the results of the job

As you can see from the fourth item in the list, there are two options. The first is to create a new table and store the information in it. This would require you to write a new datatype that would replace the current price field so that it would use the product ID in the product class to find the correct price in the new table.

Alternatively, you could update the product prices directly when the file is imported. This would involve a modification to the import routine so that it searches for the products and replaces the prices with the prices from the new import file.

Of course, you can increase this functionality by adding synchronization routines so you only upload changed products, but synchronization is a subject beyond the scope of this chapter.

Who Am I?

As part of your bridging extension, you will need to tell the external server who you are. This can be done on a trusted server basis where the user ID and password are stored within an ini file and used to authenticate you. Interestingly, this is how eZ publish works when it uses authenticated SMTP and also when it talks to your database server. Review `site.ini` to see the default settings.

Authentication

Before you read the authentication and security sections, bear in mind that millions of people send their user IDs and passwords over the Internet every day when they connect to e-mail servers via POP3 and don't give it a second thought.

These e-mail accounts probably contain just as sensitive information as that in accounting systems, but it is a risk people are willing to take for convenience.

Note that if you only receive/send encrypted e-mail or use SSL to talk to your mail server account, your connection is secure.

Your development budget should include security expenses. Don't take security for granted just because other people may ignore it.

If you want to use your eZ publish system as the main authentication point, you need to ensure that the external application can connect to eZ publish and check the status of the user. The following code can be used in an eZ publish extension to create the current user and session ID that can be passed to the external application:

```
// User id
$user =& eZUser::currentUser();
$user_id = $user->attribute( "contentobject_id" );
// Session ID
$mysessionid=eZHTTPTool::getSessionKey();
```

This information could then be used by the external application to get the user-relevant information for that user and either log in or pre-fill information.

Placing the eZ publish server into an existing environment means there is likely to be an existing, central method for managing user information. This method may be LDAP, Kerberos, Microsoft Windows domain controller, or another method.

To enable greater interoperability, eZ publish has opened up the login handler so you are able to write your own custom handlers. As mentioned at the end of the previous chapter,

these can work as a central update mechanism such as LDAP or a login replacement such as text file handlers. In either case, you will still need the user set up in the eZ publish system.

The issue here is that any authentication you perform *must* work and be secure and reliable. Now depending on the information you are securing, it can be less stringent in its checking but must still be reliable. For this reason you can use a token-based system such as **Kerberos** (http://web.mit.edu/kerberos/www/#what_is). Unfortunately, Kerberos integration with eZ publish is not yet available so it is probably better to go for a shared authentication point such as an LDAP file.

LDAP and eZ publish login methods are great, but they only allow users to log into a single system. They do not allow a user to move from system to system without having to login again. This is a problem if you have many systems that a user needs to access.

Single Sign-On

With this sign-on mechanism, upon successful authentication with eZ publish, the user's browser is presented with a cookie for the eZ publish site. Along with the cookie, the user's browser is redirected to an authentication application on the external server. This application validates the URL details sent by the user's browser and assigns the user an appropriate external single sign-on cookie, then redirects the user's browser to the eZ publish site.

The advantage of this method is that users are unaware that they have been authenticated over multiple servers, and see only a single login event.

The disadvantages of this approach are:

- Application code has to be written on the external server to provide an authenticator class to accept and validate URLs sent during redirection.
- If existing authentication methods on the external server are required to continue operating, the authenticator class must exist as a separate application and be configured to use its single sign-on facility.
- It requires cookies to be enabled on the client browser, which cannot always be guaranteed.
- A mechanism is required to synchronize the user database between eZ publish and external servers.

Variations

If a variation of the sign-on method is needed, this could take the shape of a form that is presented to the user from the external server in the following steps:

- You present a login form on your eZ publish site.
- The user submits their user ID and password on the form.

- The form posts to the external server using a form-based external authenticator.
- On success, the external application gives the users a normal external session authentication cookie. They're redirected back to eZ publish with some authentication token on the URL, which you use to initiate an eZ publish session.
- On failure, you bounce back to an eZ publish failure page.

So, there are various ways to handle authentication. In the end it is a choice that needs to be made on a project-by-project basis.

Sharing Information

When it comes to sharing information, you want to ensure that enough information is stored within your object (article, file, or even video clip) to enable it to be traced, should the system you are sharing with have a request from a user for updated or more information. An example of this is when you write a document about a particular version of software, and a reader wants to know if you have updated it for the latest version.

Fortunately, this work has already been completed by a group called the Dublin Core Metadata Initiative (<http://www.dublincore.org/>). They have a common set of items that are used by governments and companies alike to help identify the information they store. The UK Government currently uses the simple Dublin Core as the basis of its metadata standards (<http://www.e-envoy.gov.uk/Resources/Guidelines/fs/en>).

The metadata used by the UK government falls into the following sets:

Item	Usage
Creator, Date, Subject.Category, Title	Mandatory
Accessibility, Identifier, Publisher	Mandatory if applicable
Coverage, Language	Recommended

The definitions of items can be seen on the e-envoy website. Notice that all the information is either available by default, such as `Date` and `Title`, or can be added into the class definition by the developer. This shows that eZ publish can quite happily live in the public sector and fulfill the requirements that governments have for data definition.

This information could be made available via a SOAP interface and in the HTML meta tags at the top of the document. This enables you to correctly mark up information to be shared with other systems.

Communicating with Google

SOAP is a protocol that provides a standard method of communicating between applications regardless of what they are written in. This makes it perfect for us as we can communicate with anyone using the pre-built and ready-to-use tools within eZ publish.

The following example builds on the examples in the previous chapter to get information from a search engine, Google. You will use the eZ publish SOAP client to communicate with Google and retrieve the results to the search and display them in the browser.

The following example uses the eZ libraries `lib/ezsoap/classes/ezsoapclient.php` and `lib/ezsoap/classes/ezsoaprequest.php` to communicate with the Google search engine.

In order to use the following Google-based example, you will need to get your API key from <http://www.google.com/apis/>.

To see our example in action you need to have eZ publish installed and working, and then add a new virtual host to your setup, which should resemble something like this:

```
<VirtualHost *:80>
  <Directory /my/path/to/ez>
    Options FollowSymLinks Indexes ExecCGI
    AllowOverride None
  </Directory>

  RewriteEngine On
  RewriteRule !\.(gif|css|jpg|png|ico|js)$ /my/path/to/ez/google.php

  ServerAdmin admin@mysite.com
  DocumentRoot /my/path/to/ez
  ServerName mysite.myserver.com
</VirtualHost>
```

This example gives you that "instant fix" that you need to see something working well and is immediately useful. Save the following example as `google.php` in your project root directory.

The first few lines just create the HTML required to gather and display the request from Google.

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>My Google search</title>
</head>
<body>
<h2>Google SOAP search</h2>
<form method="get">
  <input type="text" size="40" name="q" value="<? print( $q ); ?>" />
  <input type="submit" value="Search" />
</form>
```

Next, we include the required eZ libraries for working with SOAP:

```
<?php
include_once( "lib/ezsoap/classes/ezsoapclient.php" );
include_once( "lib/ezsoap/classes/ezsoaprequest.php" );
```

The first real coding step is to create the eZ SOAP object and the eZ SOAP response object. This is to prepare the client and request object so they are ready to receive the parameters and send the request to Google:

```
// Instantiate a client object for communication with Google
$client = new eZSOAPClient( "api.google.com",
                           "/search/beta2" );
// Create a new request object
$request = new eZSOAPRequest( "doGoogleSearch",
                              "urn:GoogleSearch" );
```

Now we retrieve the query text, and provided our query is non-empty, we can begin to create our request. We load our request object with the parameters that Google needs for its search API. You can find more information on the parameters used here on the Google API website.

```
// Fetch the query text from HTTP GET
$q = $_GET['q'];
// Check for a non-empty query
if ( $q != "" )
{
    // Add parameter, key. You can get your own key from Google
    $request->addParameter( "key",
                           "add your key here from http://www.google.com/apis/" );
    // Send the query text as a parameter
    $request->addParameter( "q", $q );
    // Show the first results
    $request->addParameter( "start", 0 );
    // Show maximum 10 results at the time
    $request->addParameter( "maxResults", 10 );
    // Send misc. default parameters
    $request->addParameter( "filter", true );
    $request->addParameter( "restrict", "" );
    $request->addParameter( "safesearch", false );
    $request->addParameter( "lr", "" );
    $request->addParameter( "ie", "latin1" );
    $request->addParameter( "oe", "latin1" );
```

The next step is to send the request. The eZ SOAP libraries will take the information you have given through the addParameter() function and create an XML request that is then sent via HTTP to the Google servers:

```
// Send the request to Google
$response =& $client->send( $request );
```

If the SOAP server returns a fault, it will be picked up here and the fault code displayed:

```
// Check for SOAP fault
if ( $response->isFault() )
{
    // Print the fault
    print( "SOAP fault: " . $response->faultCode(). " - " .
        $response->faultString() . "" );
}
else
{
```

If the POST to the SOAP server is a success, we will set `$value` to the results returned by Google:

```
// All went ok, fetch the value from google.
$value =& $response->value();
```

This information is stored as an array and will need to be broken apart to display the results. This information is then formatted so that a page of human-readable HTML can be displayed.

```
// Print number of pages found
print( "<p>Pages found: " .
$value["estimatedTotalResultsCount"] .
    "</p>" );

// Print all search results
$searchResult = $value["resultElements"];
foreach ( $searchResult as $item )
{
    $snippet = $item["snippet"];
    $size = $item["cachedSize"];
    print( "<font color='green'>" . $item["title"] .
        "</font><br/>" );
    print( "$snippet<br>" );
    $url = $item["URL"];
    print( "<a href='$url'>$url</a> $size<br/><br/>" );
}
}
?>
</body>
</html>
```

If all has gone well with your example, the following page will be served:



Modifying Existing Code

You can truly integrate eZ publish with your application by making your application a native eZ publish extension. This means that it will need to be in PHP, but it is the best way to go and even better if it is a new project. You can use all the wrapper features of eZ publish such as authentication, content management, caching and so on, so all you do is concentrate on the core functionality of your application itself.

Summary

In this chapter, we looked at various methods of extending eZ publish. You can increase functionality via routines you have written yourself. You can integrate into the very core of eZ publish, enabling you to make your eZ publish site everything you want it to be.

The world is changing and not every site can offer the services and requirements for all users. For this reason, integrating seamlessly with external applications is a must. This ability is given to us via the SOAP. Using eZ publish, you can communicate with anyone using a common SOAP interface. We showed eZ credentials in the Google example where you are able to retrieve search results and use them as you wish. However, this is just the beginning, as more and more services come online you will be able to increase the functionality of your site in ways that were previously impossible. One example of this is to look at the number of retailers on the web that are starting to open up their environment to other sites so that all may benefit. One example of this is Amazon, which is opening up its services via a SOAP interface. Using eZ publish you are well placed to benefit from these technologies as they mature.

7

Deploying eZ publish

In this chapter, we will look at some of the strategies and techniques useful for deploying eZ publish projects. This chapter is not designed to explain *how to install a server*, but instead focuses on *what eZ publish needs in order to work well in your deployment environment*. Installing a server is a large subject in its own right and cannot be covered here in one chapter.

Although eZ publish runs quite well on *BSD, Macintosh, and MS Windows environments, it is most widely deployed in its Linux flavor. For this reason, we will focus on deploying eZ publish to a Linux environment.

This chapter will explain how to modify your server environment for it to work well with eZ publish 3 and greater.

Define Your Hosting Requirements

Before you deploy an eZ publish solution, make a note of all the hosting requirements. It is worth spending a little time at the start of development to ensure that problems do not occur at the deployment stage.

Unforeseen problems occurring at deployment equate directly to increased costs in your project.

Described below are a number of areas that need to be reviewed to ensure that you have a good picture of the deployment needs, prior to parting with your money for a hosting environment.

Number of Visitors

How many users will be visiting your site? You need to examine your current statistics or marketing plan to get a rough idea of whether hundreds, thousands, or millions of unique visitors will be visiting your site every month. At the very least, you need to make an

educated guess at this number, as the power required of the hosting server depends on this, and will indeed decide whether you need more than one machine to support your user base.

The next task is to find out how often they will be visiting—whether they visit once an hour, day, week, or month. It is good practice to overestimate this figure to ensure that your site visitors get a good quality of service. Remember that traffic will not always be evenly distributed, and is likely to peak to match the site's marketing initiatives.

User requirements	
Users per hour	
Concurrent users	
Total users	

This information will be useful if you intend to enable user accounts. The more users you have, the more disk space you will need to store their information. eZ publish 3 test installations take around 70MB of file system space and 2MB of database space. Add to this the amount of space you will need for caching page and image files and the required space soon adds up!

System requirements	
File system size	
Database size	
Bandwidth requirements	

The price of bandwidth can be a real variable with ISPs, and cannot be avoided! You could ask your ISPs to fix the bandwidth available to your site, but then you risk disappointing your visitors. So, you will need to get a sympathetic ISP that will allow you to burst traffic to your site to allow for the occasions where lots of visitors come to your site at once. It is worth getting a firm idea of the costs that this extra bandwidth will cost you, ahead of time, to help plan for these additional costs.

Security Needs

You should review the information that you will store on the site and decide on what level of security is required to protect this information. Generally, the more security you require, the more the system will cost to install and maintain.

In an ideal world, you could ensure that your site is secured to the best of the ISP's or your ability. In the real world, budgets dictate the security levels on a site. The more money you have, the better is the system administrator and time made available to you.

The good thing is that you can choose the level of security according to the type of content you are going to host.

The following content types are listed according to their typical level of importance:

1. Credit-card/Medical details
2. Personal details
3. Articles
4. News

Credit-card details usually constitute the most valuable content, but this is not necessarily true. For example, the news section could contain news for a product launch that is confidential, and if released could cause serious problems. For this reason, you need to evaluate each type of content from a company perspective to protect yourself adequately, and then budget according to that level of the risk.

The more sensitive the information stored, the more secure you need to make it.

There may already be guidelines on the level of security you need to implement. Take the time to review the requirements of your country so you do not break any local laws.

Once you have decided on the security mechanism, you need to review the type of access required to the system. In this area, there are generally three types:

- **Internet:** Open to all
- **Extranet:** Open to authenticated users via the Internet
- **Intranet (Closed system):** Open only to internal local users

As the phenomenon of individuals working from home and remote offices is now widespread, few business applications can be seen as truly Intranet-only applications, so we are actually talking about authentication and authorization for our sites.

For unprotected files, you only need to ensure that the information is valid and only updated by authorized editors.

For protected information, you need to ensure that only authorized users can edit and see the files. For this you need to use the eZ publish authentication system by assigning roles for people to edit and view the information.

To increase the level of security within eZ publish:

- Ensure that the information sent to the server cannot be read by a third party. This is normally achieved using **Secure Socket Layer (SSL)**.
- Ensure that the users are who they say they are. The user name and password are already set up, but if this is not sufficient, you need to implement a digital identity system such as a **Public Key Infrastructure (PKI)** that will help identify a user wishing to log in to the eZ system.
- Implement MD5 or SHA security to transmit the password when logging in to the eZ publish site. This will make it harder for an intruder to sniff passwords.

Reporting Requirements

Decide on the level of reporting you want from the server. Fortunately, eZ publish provides you with information about the content searches that have been made on your site. For any further information, you need to use third-party tools provided by your web server or ISP.

The following information is usually required:

- **Web statistics:** Number of users
- **Server performance:** Speed of machine, number of concurrent users
- **Bandwidth usage:** Amount of bandwidth your site uses

Web statistics are probably the most common and widespread requirement for reports, as these enable you to gather information about how many times your pages are being accessed, and by whom. As the usability of the reports created varies from package to package, it is worth reviewing the package used to create the statistics to ensure that it meets your needs.

Some common open source web access analyzer packages are:

- **Awstats** (<http://awstats.sourceforge.net>)
- **Webalizer** (<http://www.mrunix.net/webalizer>)
- **Analog** (<http://analog.sourceforge.net>)

Budget

You will probably have a good idea of the budget available for hosting, but the general rule is that the more you want, the more you pay. Although the hardware and support costs will be fixed, it is unlikely that the bandwidth costs will be. Always reserve a portion of your budget for bandwidth costs as they can rocket on a highly popular site.

Time Limits

Define the amount of time you have to prepare the deployment environment and the amount of time you have to deploy the finished system. You can use the following table:

Time	
Live date	
Time to deploy	
Deployment environment ready date	

Shared or Dedicated

Once you have defined your needs, you need to figure out whether they will be best fulfilled by a dedicated server environment or a shared one. There is a large price difference between a shared and a dedicated hosting service; a shared environment is much more economical.

Whether you use a shared or dedicated environment, you will need to ensure that the environment has the correct software available. The general rule of thumb is that if your deployment environment has Apache/IIS, PHP, and MySQL/PostgreSQL running on it already, you will probably be able to run eZ publish 3 without problems. That said, some shared environments have peculiar setups that have been created to provide extra security; these can sometimes include routines that interfere with eZ publish's smooth running. In this situation you may find that eZ publish does not run properly.

It is worth running a check on the shared server before parting with your hard earned money to be safe. You can use the eZ publish Setup Wizard that we saw in the first chapter to run a quick test. The Setup Wizard will take you through the whole process of getting a test eZ publish site ready.

If the setup screen does not appear, you will need to change the `site.ini` setting.

```
[SiteAccessSettings]
CheckValidity=true
```

This will cause eZ to run the setup test the next time you point your browser at it. You can test most functions of eZ publish by choosing one of the inbuilt demo sites.

From an eZ publish point of view, you generally only need a web interface to complete all of the operations needed. This means you can survive with a shared service that does not give you command-line access via SSH, as long as you have FTP and HTTP access. The following is a quick checklist for a shared service:

- Speed of service
- Number of sites you share the server with
- Cost of extra bandwidth

eZ publish can take its toll on a server, so ensure that the shared server has enough power for all the eZ publish sites it is hosting.

Is My Server Powerful Enough?

Whether you go for a dedicated or a shared server, you need to ensure that the server is powerful enough for your needs. Generally, eZ publish is heavy on CPU and memory usage, so it is worth ensuring your server has enough memory and CPU power for the number of users you envisage visiting your site.

You can test the system by running tests based on the number of users and the amount of information you will be storing as described in the *Define Your Hosting Requirements* section.

Even if you choose a dedicated solution, some ISPs will dictate the type of server you are able to use. If they do, ensure that they give you statistics for the server, including a test of how it works under load, running an eZ publish site. Ask your hardware provider the following questions:

- **Model name:** What server are they selling? Is it a known brand? If not, details about spares and repairs.
- **Hard disk:** What is the size of the disks; what RAID do they use? What is the space available for the database and for data? How fast are the disks?
- **CPU:** How many CPUs does the server use and what are their speeds?
- **UPS:** What is the battery life? Will the UPS be pre-configured for you?
- **Operating system:** Is this configured so you can gain SSH access?
- **Upgrade paths:** What are the options and prices, if you need to increase the number of machines or machine power for your site?

One way to measure the server's capabilities is to run tests. A good testing tool is **Siege**, which enables you to run speed tests for concurrent users. Try loading up a test site and run Siege increasing the number of current users until the server slows to a crawl. You will then be able to find the limits of your server.

Find out more about Siege from <http://www.joedog.org/siege/index.php>.

Using Siege is straightforward; the following example simulates two users hitting a dummy site for an hour:

```
>siege -uhttp://www.example.com -v -c2 -t1h -l
```

This example tests the server for 25 concurrent users:

```
>siege -uhttp://www.example.com -v -c25 -t1h -l
```

To ensure that the siege tests run correctly, you will need to run the test from a box on the same LAN, otherwise you will be limited by the bandwidth available on your Internet connection.

If you are testing on a live site, this will completely disrupt your stats for the site. It may be worth changing your statistics log location for this test.

Information about eZ publish tests can be found at:

http://ez.no/community/news/ez_publish_3_enterprise_setup_test

Documentation

Ensure that you have full and up-to-date documentation for the server, so that you have all the information you need to maintain or recover information in case of a disaster.

We will now discuss some of the areas to cover in your documentation.

How and When to Update the Documentation

This document will be a living document, so a guide on how to update it and where it is stored is a must. This information ensures that everybody who needs to update the documentation understands their responsibilities.

Contact Details

In the event of a problem, you will need to know who to contact, so that valuable minutes are not lost trying to find the number and the name of the right person at your ISP or support company.

Details should include whom you need to contact for each part of the system: your ISP, hardware supplier, and support personnel.

Location

This is the address and physical location of your where your hardware is located. It can be useful to take a photograph of your server too, for easy identification should the need arise.

Hardware

This is a complete specification of the hardware, CPU, memory, and hard disk configuration. You will need this during replacements and upgrades.

Operating System

This is the operating system your server is running, and its version.

Software

This is a complete list of the software installed on your server, with the license details. You should also note the location of any CDs or required installation media.

Patching Process

A log of the patches with the name of the administrator who applied them should be kept. This will help you rebuild the server (in the event of a failure), or troubleshoot (should things not work), as you can identify the last patch applied to the server. Some distribution providers, such as RedHat, provide tools that keep this information for you.

DNS Information

This is a list of the URLs used by the site with their expiry or renewal dates. It is worth keeping any IP mapping from DNS to IP in this section as well.

TCP/IP Information

Keep a log of all the IP addresses used by the machine. You can retrieve some of this information by running the `ifconfig` command as root. It is worth asking your ISP or administrator for a network map of your environment so you can see how their environment fits together.

Access Control

A list of all the authorized users for your system, along with their system privileges should be kept. This should be reviewed at regular intervals, especially when there are personnel changes.

Upgrade Roadmap

If you hate surprises, you should create scenarios where traffic to your site grows and you require extra bandwidth, CPU power, or memory. Use these projections to plan your upgrade path and agree on prices with your ISP. This way, you will get a better price than if you wait until you have no time to negotiate!

Disaster Recovery

Create a detailed plan of how you will recover the server or service should there be a problem. This plan should be printed out and stored at your current location and at an off-site location. Inform the people responsible for it of its location and keep it updated.

Preparing the Linux Environment

Installing eZ publish on Linux is like cooking a good pizza. If the base is firm and well formed, you are unlikely to get many problems. For this reason, it is worth spending time to choose the flavor of *NIX you should work on.

eZ publish is supported on the Linux, Solaris, HP-UX, and FreeBSD UNIX systems. Other *NIX systems may work as well. In addition, eZ publish has been tested by the community on Mandrake 9.x and RedHat ES U1.

For more information on available Linux distributions, take a look at the Distrowatch website at <http://www.distrowatch.com/>.

eZ publish comes with complete install solutions for RedHat 7.3, 8.0, 9.0, and FreeBSD to get your test environment up and running.

Although these installers will get your test environment going, **do not** use them for deploying your live site: they are not designed for this and will give you problems.

If you currently use one of the full system installers, you should review all of the security and installation settings for the packages it installs, to ensure they will run efficiently and securely.

Given that eZ publish runs on most Linux bases, it is worth finding a good base that fulfils your needs and will be supported for patches and security fixes. You will not want to be changing the environment every six months unless you have plenty of time and money on your hands. This is relevant, as RedHat is removing support for RedHat 7, 8, and 9, so these are not good choices unless you want to pay for its enterprise systems.

It is worth configuring eZ publish locally before you roll out to a live deployment box. This will let you know before you part with any money if you have the correct skills to deploy a live server.

There is one simple rule when installing and configuring deployment servers:

If in doubt, don't go live.

Installing on an Internet-accessible box is very different from installing on a local area network box, as the Internet box will get compromised extremely quickly. There are lots of *NIX security books available to help you, but you may want to run some local tests such as Tripwire or root kit checkers (http://www.chkrootkit.org/#related_links) to help achieve some peace of mind.

If you have a distribution that has all the components you need supplied as packages, you will not need to compile any packages. However, if you are working from a distribution that does not meet all your needs, you may need to compile one or all of your packages. Compiling does have its advantages, and enables you to ensure the application is tuned for your environment.

In this section we will walk through the main parts of the environment, and see how to configure them appropriately for a production eZ environment.

Apache

Apache is the web server of choice for the Linux platform and is at the time of writing going through a transition from version 1 to version 2. We will deal with version 1 as version 2 has not yet been approved for production purposes by the PHP development team. There are reports of eZ publish working well with Apache 2, but until it is off the experimental list from PHP, it is a risk.

For more information on the PHP and Apache 2 situation, take a look at <http://www.php.net/manual/en/install.apache2.php>.

If your distribution does not have the version of Apache you need, you can compile from source. This can be done in two ways. In either case you should uninstall the current

version of Apache before you continue. The first and long winded way is to define each of the modules you need:

```
./configure --sysconfdir=/etc/httpd/conf --prefix=/usr/lib/apache --
enable-module=env --enable-shared=env --enable-module=userdir --
enable-shared=userdir --enable-module=usertrack --enable-
shared=usertrack --enable-module=auth_anon --enable-shared=auth_anon
--enable-module=auth --enable-shared=auth --enable-module=alias --
enable-shared=alias --enable-module=vhost_alias --enable-
shared=vhost_alias --enable-module=rewrite --enable-shared=rewrite --
enable-module=speling --enable-shared=speling --enable-
module=log_config --enable-shared=log_config --enable-
module=log_agent --enable-shared=log_agent --enable-
module=log_referer --enable-shared=log_referer --enable-module=dir --
enable-shared=dir --enable-module=access --enable-shared=access --
enable-module=so --enable-module=ssl --enable-shared=ssl --enable-
rule=EAPI
```

The second and shorter way is to compile all of them:

```
./configure --sysconfdir=/etc/httpd/conf --prefix=/usr/lib/apache --
enable-module=all --enable-shared=max --enable-rule=EAPI
```

Once you have run the configure command, you need to compile and install the package using the following commands:

```
make
make install
```

Now that the package is installed, it is a good idea to check the installation with a simple HTML (not PHP) page. Once you have confirmed that Apache works, you are ready to install PHP.

You will need to configure your `httpd.conf` file to ensure that Apache is set up to view the file. This is not covered here. For more information on installing Apache, visit <http://httpd.apache.org/docs/install.html>.

PHP

You may well need to compile PHP, as problems have been reported using eZ publish with PHP versions below 4.2.3. The good news is that compiling PHP is very simple. The following configure command contains the features you will need for most eZ publish sites:

```
./configure --with-apxs=/usr/lib/apache/bin/apxs --with-
mysql=shared,/usr --with-ttf --with-gd --enable-gd-native-ttf --
enable-trans-sid --enable-inline-optimization --includedir=/usr --
with-zlib=/usr --with-layout=GNU --prefix=/usr --exec-prefix=/usr --
bindir=/usr/bin --with-openssl --with-xml --with-config-file-
path=/etc --with-layout=GNU --enable-mbstring --with-jpeg-dir
```

Once the configure command has successfully run, you need to reissue the following commands:

```
>make
>make install
```

The configure string contains all the options required for your standard eZ publish site. If you require more, you can, of course, modify the compilation.

If you are compiling PHP, consider joining the QA program (<http://qa.php.net/>), as this will not only help the PHP project but also help you find out if there are any problems with your environment. It will also help you fix any bugs you may encounter. Visit <http://uk.php.net/manual/en/installation.php> for more information on installing PHP.

Database (MySQL/PostgreSQL)

eZ publish works with both MySQL and PostgreSQL. If you need to compile from source, you can find more information on how to do so on their sites. However, it is preferable to use the RPMs provided by these providers.

In this section we will review how to set up the most commonly used database for eZ publish: MySQL.

The first step is to install MySQL from source or via the RPMs from the MySQL site. Once MySQL is installed, you can create an empty MySQL database via the following command:

```
>mysqladmin -uroot -p create test
```

Then access the MySQL database and grant access:

```
>mysql -uroot -p
mysql>grant all on test.* to myuser@localhost identified by 'xxxxx';
```

Now that the environment is ready, you can restore the data from your backup:

```
>mysql -umyuser -p test < test-backup.sql
```

It is tempting to store the password on the command line: `mysql -uroot -pgod`—don't do this. If you do, all your passwords will be faithfully stored in `.bash_history`, and any user with access could run:

```
>sudo cat /home/myuser/.bash_history > /home/baduser/myusers-history
```

to gain access to the root MySQL password, or run:

```
>history | grep mysqladmin
```

if your machine is left unattended.

If you require your site to be UTF-8 (UNICODE) compatible, you will need to convert the character set used by the MyISAM files. To do this, you need to shut down the MySQL database and run:

```
>myisamchk -r -q --set-character-set=utf-8  
/path/to/mysql_data_files/mydatabase/ez*.MYI
```

This database is now ready for UTF-8 data when you install the eZ publish data.

For more information on MySQL and PostgreSQL, visit <http://www.mysql.com/> and <http://www.postgresql.com/> respectively.

Two graphics engines can be used in eZ publish: GD and ImageMagick. These have been covered in Chapter 1.

GD Graphics library

The advantage of using GD is that since PHP 4.3.0, GD has been packaged directly into the PHP source code, making it always available. For more information on GD, visit <http://www.boutell.com/gd/>.

ImageMagick

ImageMagick is a command-line graphics engine to manipulate images. It can normally be found as part of the big Linux distributions and is available on Windows. For more information on ImageMagick, visit <http://www.imagemagick.com/>.

Cron Jobs

eZ publish 3 has a variety of jobs that need to be run outside of normal operations.

These include:

- Workflow
- Notification
- Linkcheck
- Unpublish
- RSS import

Although these routines can be run from the admin interface, it is usually more convenient to run them via a cron job.

Before you let the cron job loose, it's worth checking the job first from the command line.

```
>cd /path/to/ez/project  
>php -C runcronjobs.php
```

This will let you know that the routine runs correctly. If it does, you will now need to automate the process using a script run from your cron engine. For most distributions this is in `/etc/cron.daily/`. You will need to create a script following the conventions of your Linux distribution. In our example, we will use `S99_ezpublish_cron.sh` as we want this task to be one of the last tasks run in the daily routine.

```
#!/bin/sh
cd /path/to/ez/project/;
php -C runcronjobs.php 1>/log/ez/ez_cron
```

This will work, but it runs the eZ routine as root, so it is advised that you run as another user with less permissions. In this example we have created a user called `ezuser`.

```
#!/bin/sh
su -l --command='cd /path/to/ez/project/; php -C runcronjobs.php
1>/log/ez/ez_cron' ezuser
```

Running the eZ routine as this user lessens its authority and reduces the security risk of having maliciously modified code having root access to your system. You could also create a user cron file and store the jobs to run in `/var/spool/cron/crontabs/<user>`.

SMTP

If you want to send e-mail from the server rather than via a remote SMTP server, you will need to install a **Message Transfer Agent (MTA)** such as **Sendmail** or **Postfix** to ensure that eZ publish can send its e-mails. To test if this is working, run:

```
mail myemail@example.com
Subject: test for eZ
testing eZ publish mail
Cc:
```

If you receive this mail, the mail functionality will work with eZ publish.

PHP Accelerators

eZ publish has been designed to utilize PHP accelerators to speed up its execution. By using an accelerator, you will see as much as a 60%-80% speed increase over a non-accelerated site.

It does seem like a no-brainer, but it is worth picking your accelerator carefully. You may find that some do not work fully with your entire site with eZ publish extensions or may have an upper limit to the number of pages they can successfully cache.

Some cache engines have been known to fail when you have tens of thousands of cached files, so test with a fully laden site or sites.

Several accelerators work with eZ publish:

- **Turck MMCache** (<http://turck-mmcache.sourceforge.net/>)
- **ionCube accelerator** (<http://www.phpaccelerator.co.uk/>)
- **APC Alternative PHP Cache** (<http://apc.communityconnect.com/>)
- **Zend performance Suite** (<http://www.zend.com/store/products/zend-performance-suite.php>)

Installation instructions for each of these packages are provided on their respective websites.

Deploying

Now that the production environment is up and running, it is time to move your project from your local development and testing environment to the deployment site.

It is worth going through the checklist presented later to see if you are ready. It is an example of the general items needed. There are a few things you need to change when moving from development to deployment, so it's worth having a list so you don't miss any. After all, you don't want live orders going to your test mailbox, do you?

It is a common misconception that the binary installation packages supplied with the eZ publish Setup Wizard are ready to go from a deployment point of view—this is simply not the case. These installers are only meant as an example of use and require tweaking if they are to be used in a live environment.

Ensure that all the default settings match your needs. Check all the `.ini` files and templates for areas that could give your site problems from an embarrassment or security point of view.

Run through the following deployment checklist:

- **URL mapping:** If you are using `map` as your `HostMatchType`, you will need to change these for the production values.
- **E-mail addresses:** Review your `.ini` files and classes for any test e-mail addresses that have been used and change them to the production ones.
- **ImageMagick convert location:** Ensure that the `convert` binary location is set for the deployment box in your `image.ini` override.
- **File permissions:** Ensure that you set the file permissions to prevent world-access to your files.
- **Users and roles:** Ensure that no test user names and passwords are still present. Also ensure that any default passwords/permissions are changed to

the production settings. In particular, ensure that the admin password is no longer publish.

- **Debug settings:** Make sure this is turned off or set to an IP-specific location.
- **Cache settings:** Ensure that this is turned on.
- **Images:** If you have changed the database and uploaded new images, grab a copy of var/storage and place it on the server. If you have not done this, eZ publish will try to display images that are not in your deployment file system.
- **Database:** Grab a snapshot of the prepared database ready for the deployment box.

Once you have gone through this checklist, you are ready to upload the project to the production environment.

New Project Deployment

It is easier to deploy a new project than it is to update an old one. With a new project, you only need to take a copy of the entire project base with its database, and upload it to the deployment environment.

Updating Project Deployment

When updating a project, when you are dealing with existing data, you need to ensure the previous site is running right up until the point of your new release.

First, decide whether your update requires changes to the database and how extensive they are. If you deem it necessary, you could freeze content changes for a period of time before your update, to give yourself a chance to make updates to the system.

Of course your project may be different, especially if bringing the database to your location is not an option. In this case you will need to make other plans. The following example gives a general outline of the tasks involved.

Here is a quick script for changing permissions on an eZ publish directory:

```
#!/bin/sh
# Prepare project directory after update for current directory
ldir="."
cd $ldir
echo "start"
# Check to see if we are in the right directory
if [ -d kernel ]; then
    echo "directory exists"
else
    echo "directory does not exists, ending"
    pwd
    exit

```

```

fi
echo "clean up permissions"
chown -R apache.ezpublish *
chown -R apache.apache var

# ensure that apache user "apache" does not have write access to the
files
# ensure group has full access to files
chmod -R 570 *
chmod -R 770 var
echo "complete"

```

Backups

To ensure that your system is backed up, you will need to back up the change data and file locations. This entails backing up the var directory tree in your project directory and the database.

This can be achieved via a cron job, by running a tarball for the changing files daily:

```
>tar cvzf /backup/myproject-daily-ddmmyyyy.tgz
/path/to/my/project/var
```

or monthly:

```
>tar cvzf /backup/myproject-full-ddmmyyyy.tgz /path/to/my/project
```

and a dump for the database:

```
>mysqldump -umybackup -ppassword --add-drop-table --lock-tables
--complete-insert mydb >/backup/mysql/mydb-ddmmy .sql
```

No backup is complete until it has been tested, so download the files and try to recreate the system using your backups on your local box.

To automate this project, you could use a great backup tool called **REOBACK** at <http://sourceforge.net/projects/reoback/>, or **DrakBackup**, which comes with the Mandrake 9.2 Linux distribution.

Ports

eZ publish in its basic form only needs to have port 80 available, so unless you need to run SSL (port 443), you should lock external access to port 80. You need to ensure that your server works with DNS, e-mail, and so on, so you will need to review their requirements.

To ensure that only the correct ports are open, you can use a port scanning tool. One such widely used tool is **nmap** (<http://www.insecure.org/nmap/>).

Summary

In this chapter we reviewed the main elements of deploying a site, from the analysis and requirements, to hosting, testing, and deployment. Hopefully, this will have given you an insight into what is involved in deploying your own eZ publish site.

The process of deployment and the live environment are possibly two of the highest-risk elements in any project. In an ideal world, deployment should be painless, be on time with the environment being secure and reliable, and meet the users needs for functionality, speed and flexibility.

These are high ideals, and most deployments will have some (if only minor) problems. For this reason, your deployment needs the most careful planning and execution. The goal is to be prepared and practiced in a safe environment beforehand. If you have created your plan, tested it, and followed it, most problems will be reduced to only minor issues. However, remember that your site is going into a place where not all are nice and friendly. Plan for the worst that can happen and you will not go far wrong.

Finally, some people say that running a web site is like bringing up a child. Well, if that is the case, then by deploying it correctly you have given it the best start in life. The task now is to ensure that the site receives regular attention to ensure that its growth is secure and sustained.

8

Center for Design at RMIT Case Study

So far in the book we have covered almost every aspect of eZ publish. We started with installation and configuration, moved on to adding content and understanding content types, working with templates, and understanding the core framework. The later chapters discussed how to extend the functionality of eZ publish, develop our own modules, and finally how to deploy and optimize our installation.

In this chapter, we will take an in-depth look at the implementation of an eZ publish site. This case study will take you through an entire project, from conception to deployment, explaining all the stages of the project, along with the various problems encountered at each stage and their resolution. We begin with a discussion of the client.

The Client

The Centre for Design promotes environmental sustainability through a directed program of research, consulting, professional development, and knowledge sharing. It is recognized internationally as a leader in the development of design methods and tools that support sustainable product design. Its programs focus on sustainability and eco-efficiency as a source of innovation and responsible development.

The organization is supported by the Royal Melbourne Institute of Technology (RMIT) but must make its full budget through commercial commissioned consulting work. It is based in the Faculty of the Constructed Environment at RMIT's city campus in Melbourne. RMIT is one of Australia's largest and most respected technical and design universities. The integrity of their consulting service comes from their university-based research and yet they have to be commercially realistic to survive and grow. Their revenue is from research and consulting services, training, and publishing reference material in three areas of design: sustainable products and product systems, sustainable buildings, and life cycle assessment.

By way of example, here is some of the work they have done and are doing:

- **A pilot stewardship project—Beyond the Dead TV:** Managing end-of-life consumer electronics in Victoria. The aim of the project was to develop sustainable solutions that helped reduce and ultimately eliminate hazardous materials from end-of-life electronics (e-waste) entering landfill and presenting ecological or human health problems. More than 3,500 TVs, computer monitors, and VCRs were diverted from landfill during the project. Every year, a large number of discarded electrical and electronic products enter the waste stream. There are significant opportunities to recover and reuse much of the metal, plastics, glass, and other materials. Barriers to collection, as well as expected costs for collection and disassembly operations, were studied.
- **The EcoHome:** A new model project home (the EcoHome) was built in Deer Park, in outer western Melbourne. This research project investigated the sustainability outcomes possible in outer suburban project homes using current building and design technologies, and the barriers to the uptake of these technologies more broadly in outer suburban project homes. The outcomes included a critically needed decision support tool to help the industry meet regulatory requirements for more sustainable housing.
- **Centre for Education and Research in Environmental Strategies:** RMIT is working with CERES (Centre for Education and Research in Environmental Strategies) through the Centre for Design to assist in developing an urban water conservation demonstration and research facility.

Being a cross between a commercial organization and a University's research division, the Centre for Design is treading a fine path of commercial promotion and a reference authority. This project was essentially a challenge to deliver a site that met both objectives with integrity.

Being the centre for *design* and the national leading authority on design issues for environmental sustainability, the site needed to reflect this positioning in its information structure, navigation, and visual design.

This organization's value lies in its information—the value of the information is in its depth of non-commercial research. Sharing information increases the reputation and value of the organization. A website is the perfect place to transfer knowledge globally.

As environmental design is the way of the future, the Centre and its public presentation materials had to project the intention, depth, value, and professionalism of the organization.

The Existing Site

The first website was built in 1996. At that time, most websites were built with HTML and simple graphics. This site was no exception. The navigation was implemented using a series of icons that allowed the user to click through to each section of the site. While clever, the icons had little intrinsic meaning and no text to explain where you would end up if you clicked them. However, most pages had regular hyperlinks that gave the user a better idea of what the site contained.

What was interesting about the existing site was how it had grown over time. The site was updated fairly regularly over a seven-year period by many people with varying levels of technical skill. The result was an extremely difficult-to-navigate website with a remarkable amount of content. Updating sections of the site became the responsibility of the managers that ran those areas within the business. One of the larger sections of the site, Life Cycle Assessment (LCA), was maintained by the Manager of LCA for the Centre for Design. This section of the site had internal consistency as one person was in charge of its content and maintenance. However, the structure of the content in the LCA section differed from other parts of the website. The colors, navigation, and structure were all unique to this section.

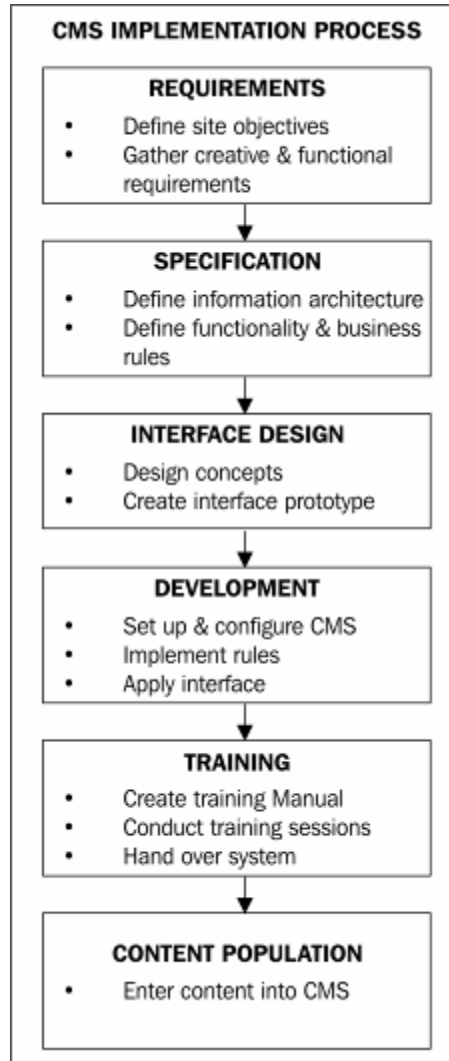
Over seven years, the existing site had become so fragmented and disjointed that even staff within the CFD had trouble finding content. A copy of the existing site was archived before deploying the new site using the eZ publish CMS to ensure that content would not be lost.

The Project

Initially, the requirements were to provide a consistent look and feel to the existing information as well as the ability to update certain sections. The focus was purely on the content and its presentation. As the need for content management was limited to only some of the sections, we initially proposed a custom-built CMS. However, we also stated that the needs of content management could go beyond expectations, so a full content management system would be a better long-term option. This proposal was presented in November 2002.

When we discussed the proposal with the client, it became clear that the needs of the new website could not be fulfilled with a simple custom-built CMS. The depth and scope of content was far too wide. A custom-built CMS might have provided an acceptable short-term solution, but considering the life span of the previous site (seven years) we knew that this site had a minimum two year life span and an expected life span of five years. Based on this, we went back to the client in December 2002 to establish the detailed requirements of the site.

The Process



Requirements

Requirements gathering was conducted over several meetings and finalized in January 2003. The requirements covered the following areas:

Key Objectives

- Promote the organization and its services
- Clearly define what CFD are about
- Create a knowledge base
- Publish information in order to change current practices

Creative

- Present CFD as they wished to be seen
- Show an understanding of the target market
- Recognize local & international considerations

Functionality

- All content to be managed in-house without needing technical skills
- Intuitive and easy-to-use browser-based administration interface
- Multiple users with different permission levels (namely author, editor, and publisher)
- Consistent presentation for all content
- Site search facility (including searching Word documents & PDFs)
- Complex publication rules (for example, content added to one section can automatically appear in other sections)
- Accommodate different types of content
- Workflow for creation, review, and publication of content
- Version tracking
- Ability to update look and feel without rebuilding the entire site

Content

- Review existing content
- Establish patterns
- Establish ideal structure

Hosting Environment

Sorting out the hardware was straightforward. We already had a server configured that hosted other eZ publish sites. The client agreed that we would host the new site.

Hardware

- Hewlett Packard LC 2000R
- Dual PIII 1GHz processors
- 1Gb Ram
- 3 x 9Gb SCSI Hot Swappable Hard Drives
- RAID 5 Array
- DAT tape back up facility with daily tape rotation
- Dual Redundant Power Supply

Software

- Red Hat Linux
- Apache web server
- Tomcat 4 servlet container
- PHP 4
- MySQL 3

Selecting a CMS

What made this project unique was the combination of the number of authors, the number of content types, and the rules for how content was to be displayed. Individually, these requirements weren't difficult, but combined, they became far more complex.

We had previously evaluated CMS solutions and found that eZ publish could immediately satisfy the majority of the requirements at face value. The only requirement not covered was searching Word and PDF documents, but this was on the roadmap for

implementation within the next 12 months. Our experience with eZ publish gave us confidence that it was the right solution for this project.

Once the requirements had been established and eZ publish had been selected as the CMS solution, we had to work out the requirements to be implemented as a part of Stage 1. The aim was to have the site live by the end of May 2002. Given this timeframe and the budget, we knew that it would not be possible to deliver all the requirements. We discussed priorities with the client to establish what had to be in Stage 1 and what could wait until later stages.

The priority was to get the site up, as the Centre was not keeping up with best practices in their field; this was having an impact on their reputation both nationally and internationally. Another priority was making sure the content was properly structured and presented. It was agreed that the implementation of workflow with multiple levels of permissions could wait until Stage 2.

Specifications

Now that we were clear what to achieve, and by when, we needed to specify exactly how we planned to make it happen. We did this by creating an information architect document that contained the following sections:

- User View
- Admin View
- Content Model
- Display Templates
- Content Types

User View

Most of the work for this was done in the initial audit. We started by defining a tree structure of what the main sections of the site were, and then defined sub-sections as well as the dynamic content.

Home

About the Centre

Article A

Article B, etc.

Sustainable Products

Article A etc.

Client A

Client B etc.

Publication A

Publication B etc.

Link A

Link B, etc.

Project A

Project B, etc.

Training A

Research & Consulting

Article A

Article B

Publications

Publication A

Publication B

Sustainable Products

Publication A, B, etc.

Sustainable Buildings

Publication A, B, etc.

Life Cycle Assessment

Publication A, B, etc.

Note: Items in italics were added by the client.

Most of the structure was straightforward. There were a number of sections, and in each section, different types of content could be stored. The complexity arose when content placed in one section was to also appear in another section. For example, a publication appearing in the main section Sustainable Products would also appear under the main section Publications in the Sustainable Products sub-section.

Although the tree structure accurately captured how the user was to view the site, we found that it was difficult for the client to visualize how this would translate into a website. It also did not clearly show that a publication shown in different sections was in fact the same content (just one key feature of a CMS). In later implementations, we

replaced the tree view with a simple sitemap, which has proved to be far more effective. To show the rules of where content was to appear, we used a content model.

Admin View

The Admin view is how people managing the content see the site. However, this is not necessarily how the user sees the site—from a display perspective as well as the structuring of information. As mentioned earlier, a user view is best represented by a sitemap; for the Admin view we found the tree structure to work well. The Admin view is about where to add, edit, and delete content. It's like a file server where the content is organized in a logical manner for easy access. How that content is viewed by the user is a different thing altogether. The purpose of the Admin view is to make things easier for the people creating and managing the content. The Admin view became a working area, a repository of information that needed to be structured appropriately.

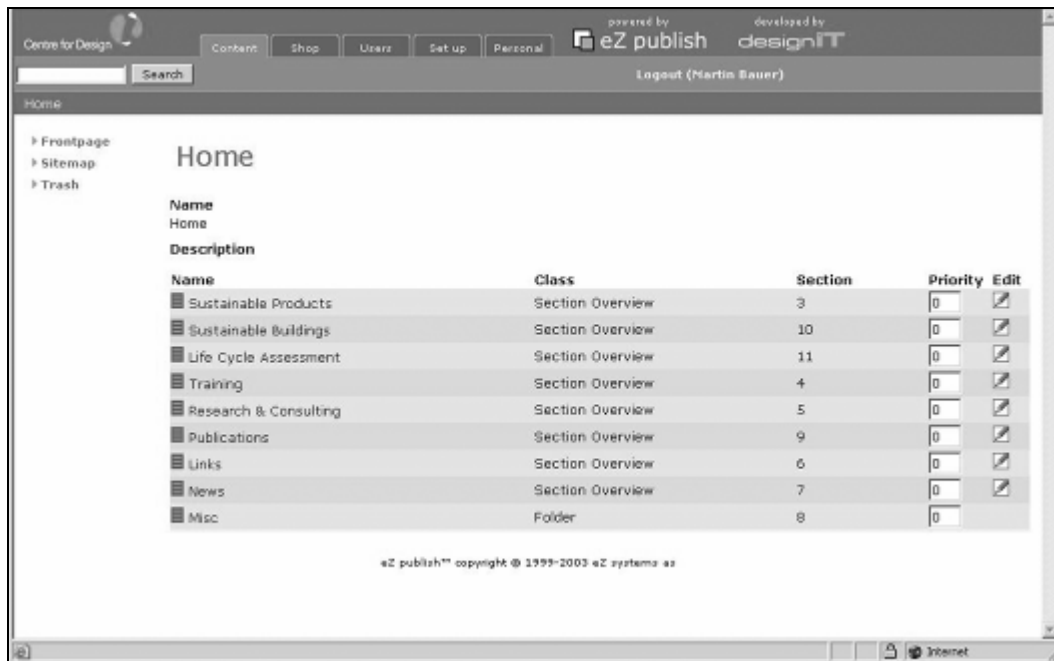
As content entered in one area was to appear in other areas, we initially proposed that the admin section be structured according to the content types. Thus when logging into the admin section, rather than getting a list of sections as defined in the user view, we proposed providing a list of content types:

Content Type	Appears In
Publications	Sustainable Products
	Sustainable Buildings
	Life Cycle Assessment
	Publications
Projects	Sustainable Products
	Sustainable Buildings
	Life Cycle Assessment
Training	Sustainable Products
	Sustainable Buildings
	Life Cycle Assessment
	Training

Although on the surface this made sense to both us and the client, the implementation did not prove to be intuitive from a usability perspective. The end decision was to implement a hybrid approach. We listed the main sections as per the user view, also listing content types that didn't have their own dedicated section, for example, Links.

The admin page would contain the following sections:

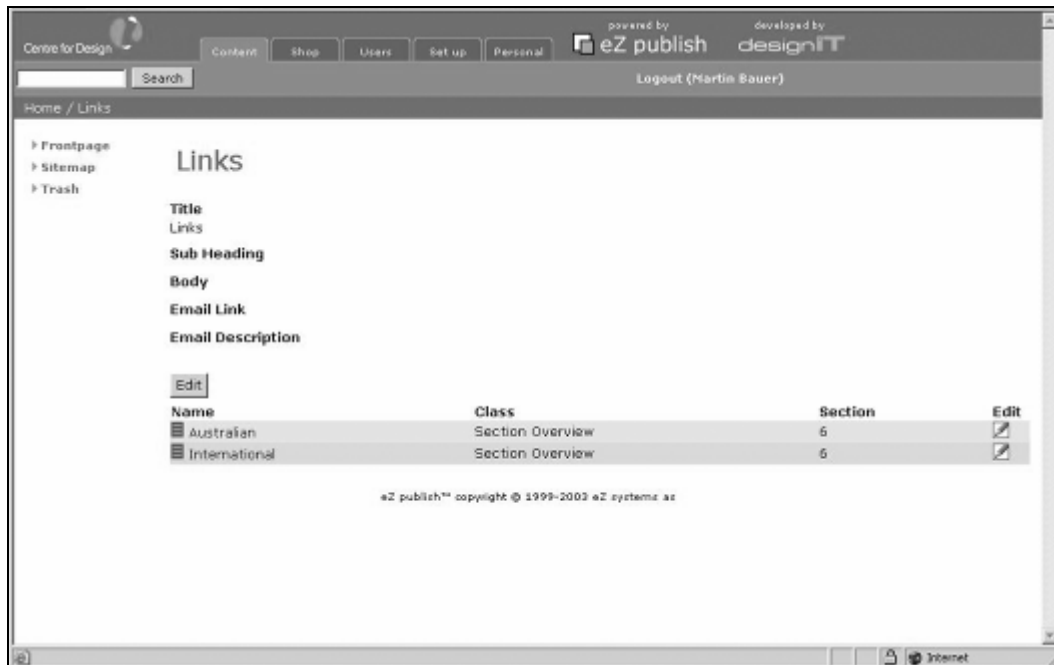
- Sustainable Products
- Sustainable Buildings
- Life Cycle Assessment (LCA)
- Training
- Research & Consulting
- Publications
- Links
- News
- Misc



Most of these sections worked in a straightforward manner: you simply clicked on the section and added content. Where content should appear was defined within the system, so the administrator didn't have to worry about adding the content in more than one place. The exceptions to this were Links and Misc.

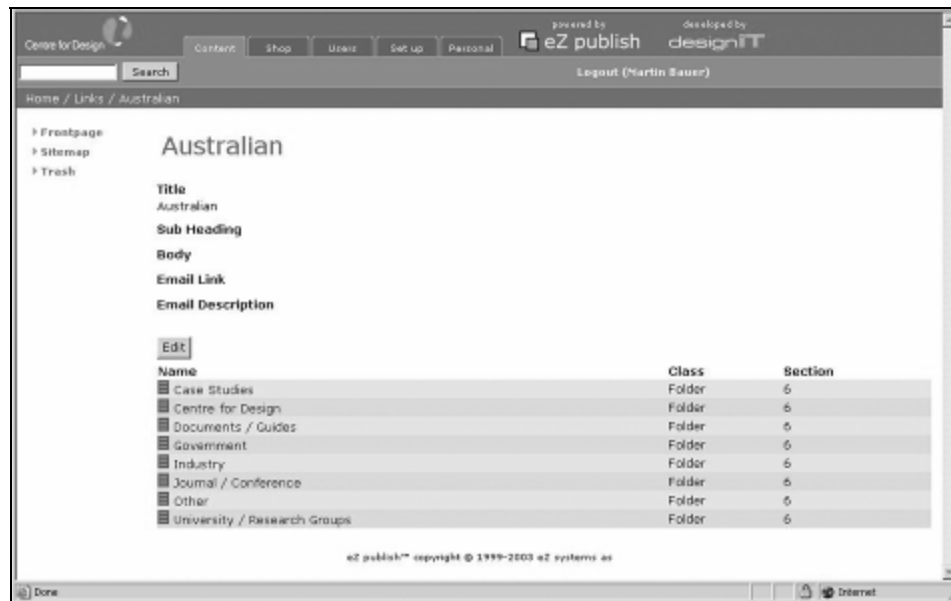
Links

In the existing site, links had great importance as a part of the knowledge base and were of great value to people using the existing site as a research tool. It was important that the new site allow for the links to have the same level of importance and to be presented in a similar way. To allow for this, we extended the directory-style approach, categorizing them into different sections and then sub-sections.



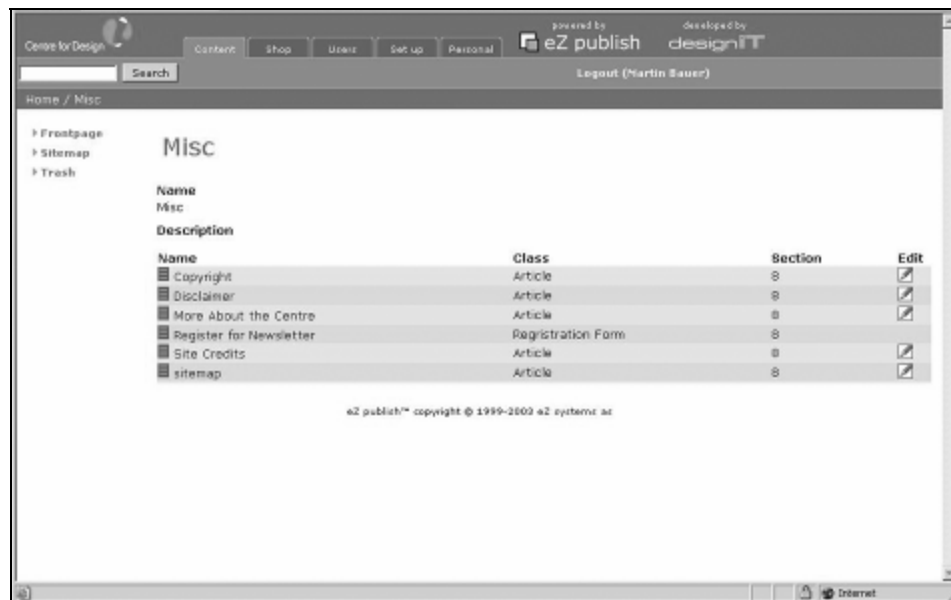
A link was either local or international, and would fall into one of the following sub-sections:

- Case Studies
- Centre for Design
- Documents/Guides
- Government
- Industry
- Journal/Conference
- Other
- University/Research Groups



Miscellaneous

This covered all the single pages within the site that are better described as utilities (*"Don't Make Me Think"*, Steve Krug, *New Riders*). In Misc, we put content for things such as the copyright message, disclaimer, sitemap, credits, and so on.



It was a repository for anything that was information about the site or the centre, rather than the information that CFD provided. Links to these items appeared in the footer on each page of the site.

In recent implementations, we have found it useful to add another section, Library. In it, we store files (such as Word documents or PDFs) or images that are used in more than one place on the site. It is especially handy for reusing graphics. We've found that some clients use the library a bit like a file server and create sub-directories to help structure the content. The inclusion of a Media view in the admin templates of the version 3.2 of eZ publish makes the management of a library of images much more user-friendly.

Content Model

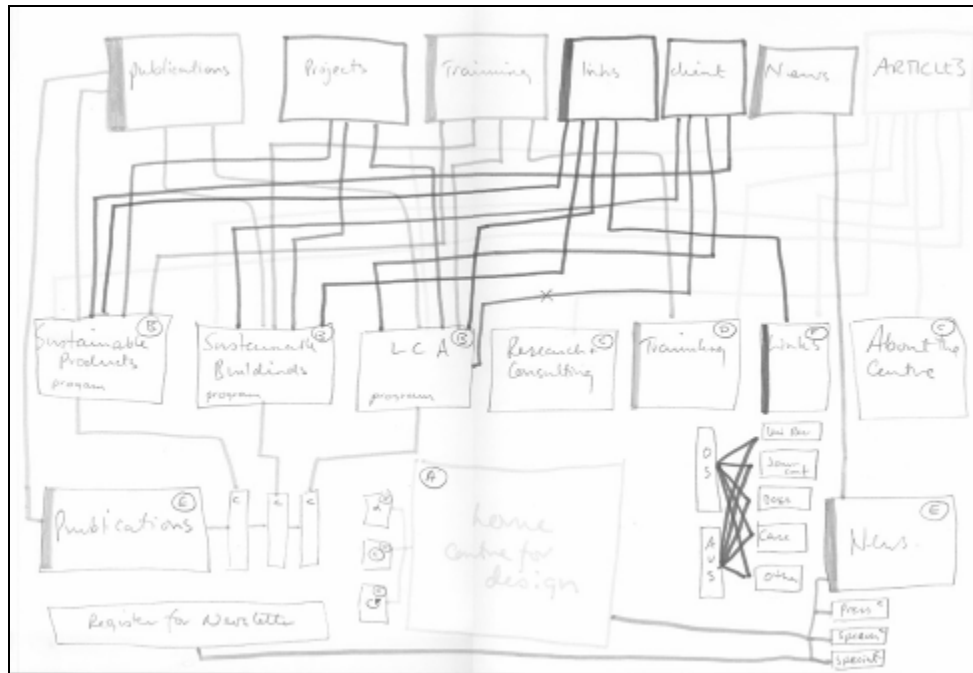
Having defined the way users would see the site and how administrators would work with the content, we had to define the business rules that would control not only what the user saw in each section, but also what the administrators would be able to add in each section. In the user view we used a sitemap and in the Admin view we used a tree structure, but neither of these approaches was effective in capturing the business rules of what content could go where. It was important to capture the rules in a manner that had enough detail for the developer to work with but was also easy enough for the client and designer to understand.

Traditionally, sites with a database are displayed using a sitemap and a line to a database. The details of the database are defined separately in a schema, which in itself has different views: for example, table definition and relationship diagram. We needed to combine the sitemap with the content types and show the relationship between them and other parts of the site.

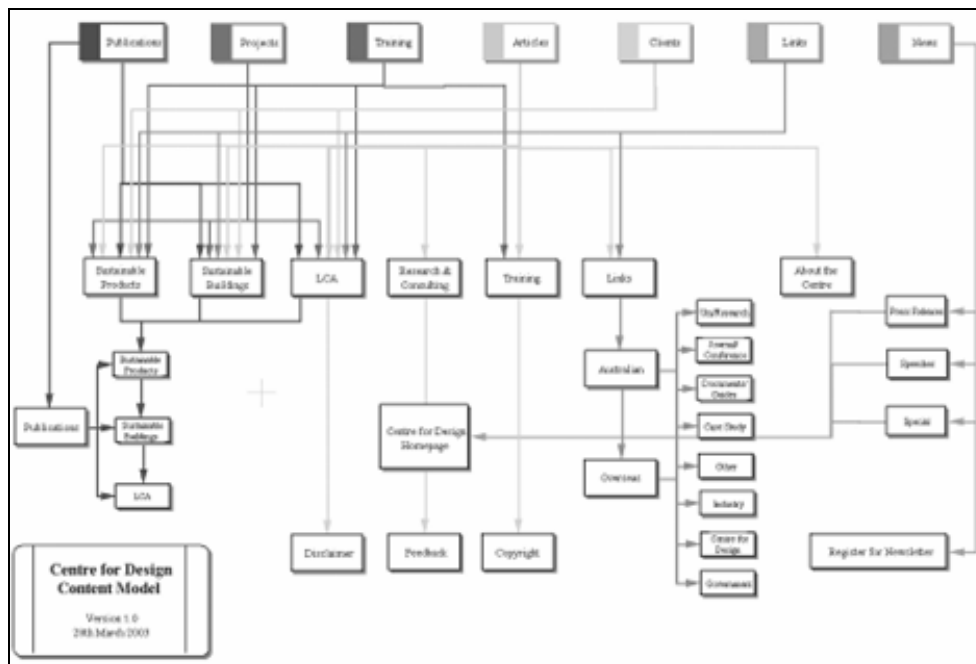
After several attempts at trying to combine the sitemap with database schema type diagrams, we decided to adopt a different approach that started with the content rather than where the content went. Adding to this, we borrowed from Peter Coad's approach to object modeling in color (*Java Modeling in Color with UML*, Peter Coad, Eric Lefebvre, Jeff De Luca, Prentice Hall). The end result is what we call a content model.

To create the content model, we start by representing each content type at the top of the page, each with a different color. We then represent each of the main sections of the site underneath. The next step is to draw a line between the content type and each section that the content type can appear in. This establishes the rules of what can go where.

Once the main rules are defined, we look to capturing when content should be published in two places simultaneously. In the following diagram, a news article stored in the speeches section is also published on the home page. A publication stored in the LCA section is also published in Publication under the LCA sub-section.



Original Content Model



Final Content Model

This approach proved to be quite powerful. It was easy for the client to understand the relationship between the content and the site. At the same time, the developer was provided with a high-level view of the relationships that would be implemented within the CMS.

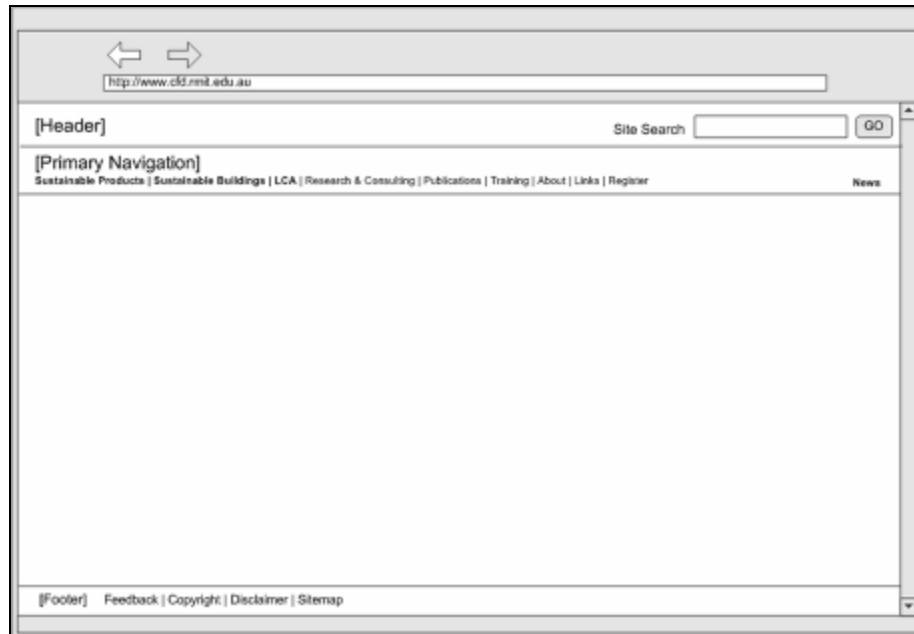
In the same way that an object model captures the objects within a business domain and how they work together, the content model brought together the content, the site, and the business rules in a single effective representation. Because the content model brings so many elements together, it is the most important part of the specification to get right.

Get the content model right and everything else will fall into place.

Display Templates

Having captured the high-level user view and the content model, the next step was to work out the details of how the navigation was to work and how content was to be displayed. We created a series of display templates (or **wireframes**) and documented all the elements that would be displayed on that page; in other words, the presentation logic.

We started with the global template and then moved on to each section and sub-section until we had captured how any and every page on the site would be displayed. The following template defines the elements that appear on all pages:



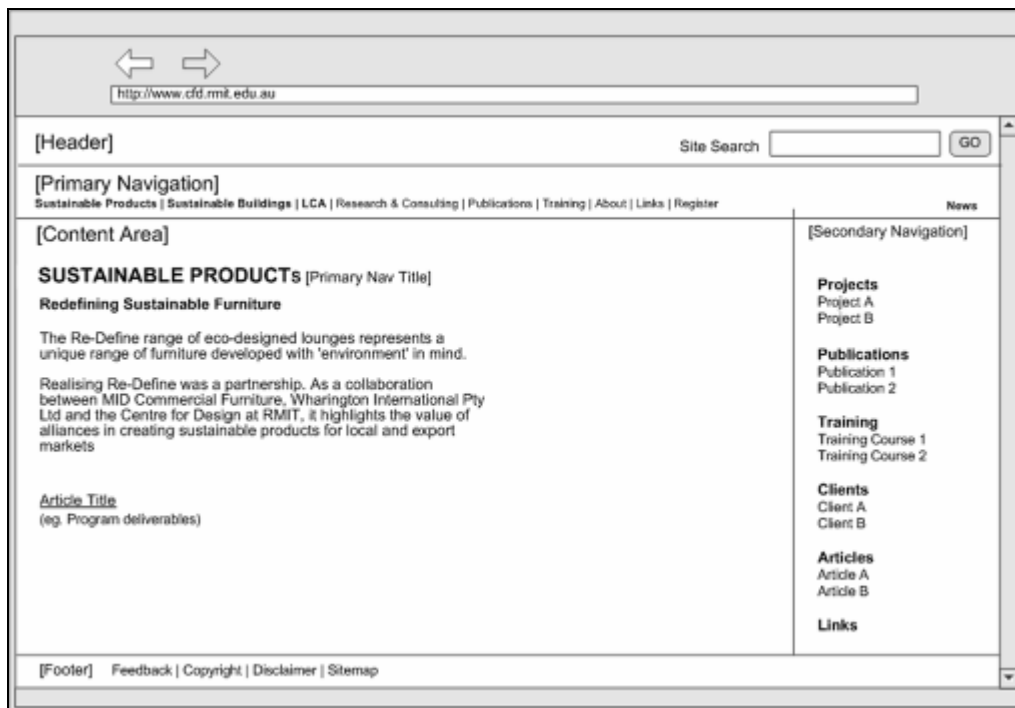
The various sections of this template were:

- **Header:** Displays logo & search window
- **Primary Navigation:** Displays all primary navigation items
- **Footer:** Displays feedback, copyright link, disclaimer, and sitemap

Once we had defined the global elements, we looked at each section to define what content was to appear in it, the sub-navigation, and the specific rules that defined how and where the content would be displayed on the page.

Sustainable Products, Sustainable Buildings, and LCA Template

This template was for the main sections of the site. All the global elements are displayed. The sub-navigation is displayed as links under the type of content that they link to: for example, links to Project A and Project B under the heading Projects:



The sections of this template are:

- **Global Elements:** Display all
- **Secondary Navigation:** Display all secondary navigation items under category headings and titles as links to full content

- **Content Area (Sustainable Products / Sustainable Buildings / LCA):**
Display selected primary navigation title, selected primary navigation overview article, and article title as link to full article

Content Types

Defining all the content types proved to be one of the bigger challenges. It was difficult to identify patterns in the existing site to use as a basis to accommodate all of the content. For example, not all training courses had the same information; some had a breakdown of the content, others didn't. It took a number of revisions to be able to consolidate all of the content into eight content types that would allow CFD to port everything on the existing site to the new site.

The content types were as follows:

- Overview Article
- Article
- Project
- Training Course
- Publication
- Link
- Client
- News

The next step was to define each content type in more depth. Borrowing from database definition and OO terminology, we defined each content type in terms of attributes and datatypes.

An attribute represented each element of a content type: for example, heading and description. The datatype represented how the information was to be stored, for example, rich text, plain text, or numeric.

We also provided an example of each attribute as a part of the definition to ensure that the name of the attribute was meaningful and the datatype was appropriate. This was a very important step.

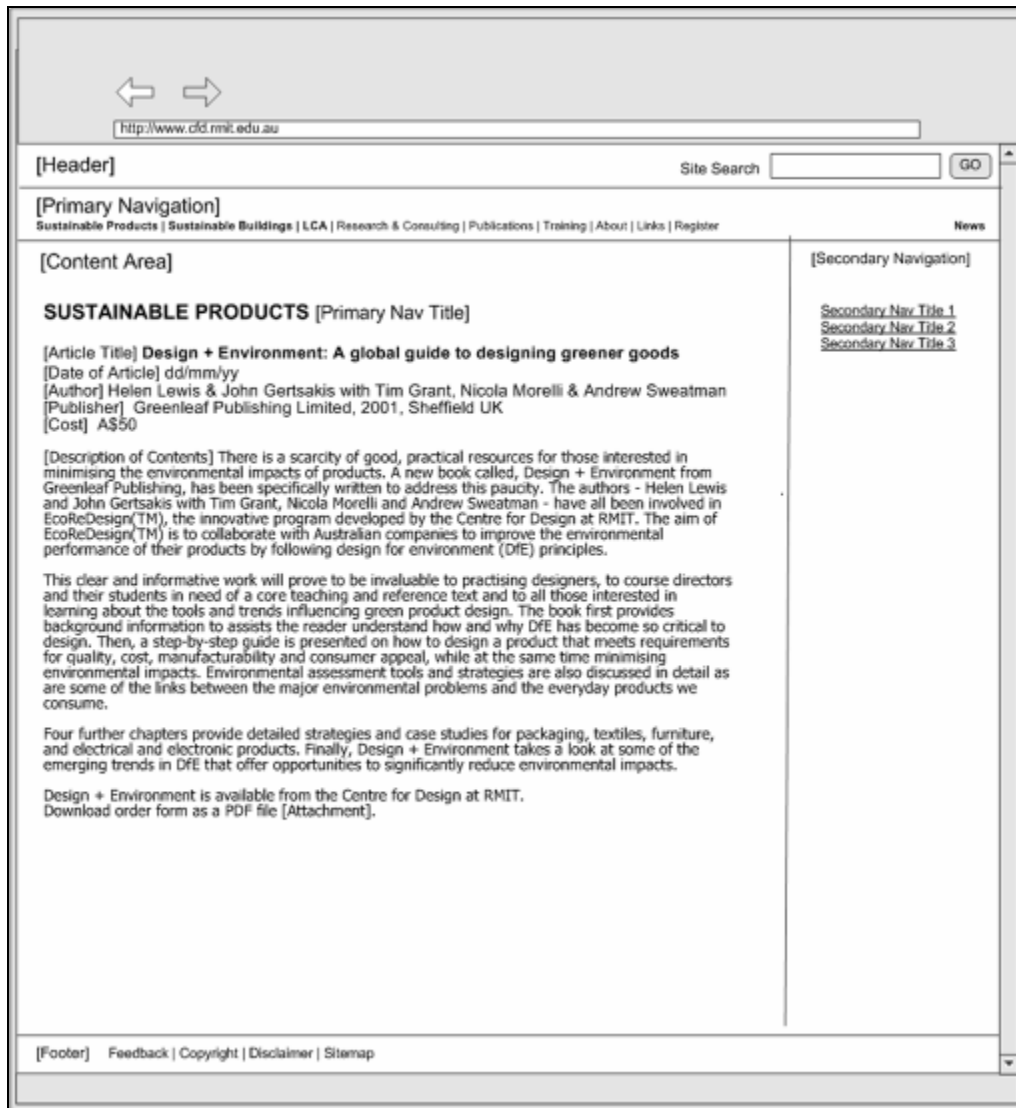
In a previous implementation, we found that although the content type definition made sense during the specification stage, the definition no longer worked when it came to adding the actual content, and changes had to be made.

Adding an example at this definition stage is a simple but effective way of checking that you are on the right track.

The end result was a table for each content type. Here is the table for the Publication content type:

Attribute	Type	Example:
Title*	Text line	Design + Environment: A global guide to designing greener goods
Author*	XML text Field	Helen Lewis & John Gertsakis with Tim Grant, Nicola Morelli & Andrew Sweatman
Date of Publication*	DateTime	No example
Publisher	Text Line	Greenleaf Publishing Limited, 2001, Sheffield UK
Image	Image	Image of front cover of publication
Description of Contents*	XML text Field	There is a scarcity of good, practical resources for those interested in minimizing the environmental impacts of products. A new book, called Design + Environment from Greenleaf Publishing, has been specifically written to address this paucity. The authors—Helen Lewis and John Gertsakis with Tim Grant, Nicola Morelli and Andrew Sweatman - have all been involved in EcoReDesign(TM), the innovative program developed by the Centre for Design at RMIT. The aim of EcoReDesign(TM) is to collaborate with Australian companies to improve the environmental performance of their products by following design for environment (DfE) principles. Download order form as a PDF file.
Cost	XML text Field	A\$50
Attachment (order form pdf)	File	
Attachment Description	Text Line	Download order form

The final step in defining the content types was to work out how the information was to be displayed. To do this, we created a sample display template for each content type that would show each attribute. This was an important step, as in some cases seeing the information laid out suggested changes to the definition of the content type that produced a better end result. It was far easier and quicker to make changes to the definition or display template at this point than after the site was built.



The process of creating the information architecture document was long and difficult. The document went through several revisions and reviews with the client. During these reviews, we went through every element of the specification until we were sure that we had captured everything correctly. After each revision, we made sure both the designer and developer also had a chance to give their input from an implementation perspective.

Overall, this process took approximately two months from start to finish; it took time to organize the reviews and coordinate getting all the right people together at the same time.

Although it was a very difficult and complex process, it proved to be a wise investment; the end result required very few changes from what was contained in the information architecture document.

Interface Design

As the client had such an outstanding reputation with world thinkers in the **environment design** sector, it was imperative that the site looked and felt like it was a *unique individual* with best-practice design sensibilities and an inspirational interface.

Visual Design

Earlier, the Centre for Design had a terrible logo – very 90's, hard, and academic. The first job was to talk the Centre into getting a good visual device, i.e. a logo, to associate their name with. We recommended a logo designer and with the brief he came up with a logo the Centre is very happy with. We then moved on to navigation.

Once we sorted the exact number, type, and informational priority of navigation items, it became easy to display them in an innovative, yet clear and logical manner.

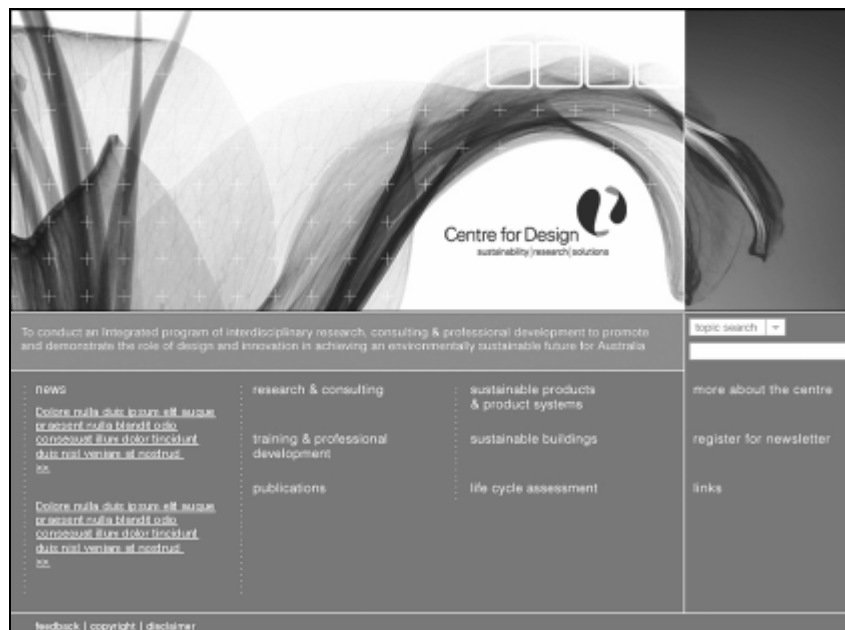
The home page design needed to have a real personality and the secondary pages needed to draw on that personality to surround the internal content without intrusion:

- The site had to say: we are modern, eco-sensitive, unique, dramatic, and 'we are where design is at'
- It had to be: non-threatening, classic, timeless, and easy to navigate
- It had to have: the Centre's purpose clearly displayed, and the navigation near that area

Our initial concepts are shown in the following screenshots:

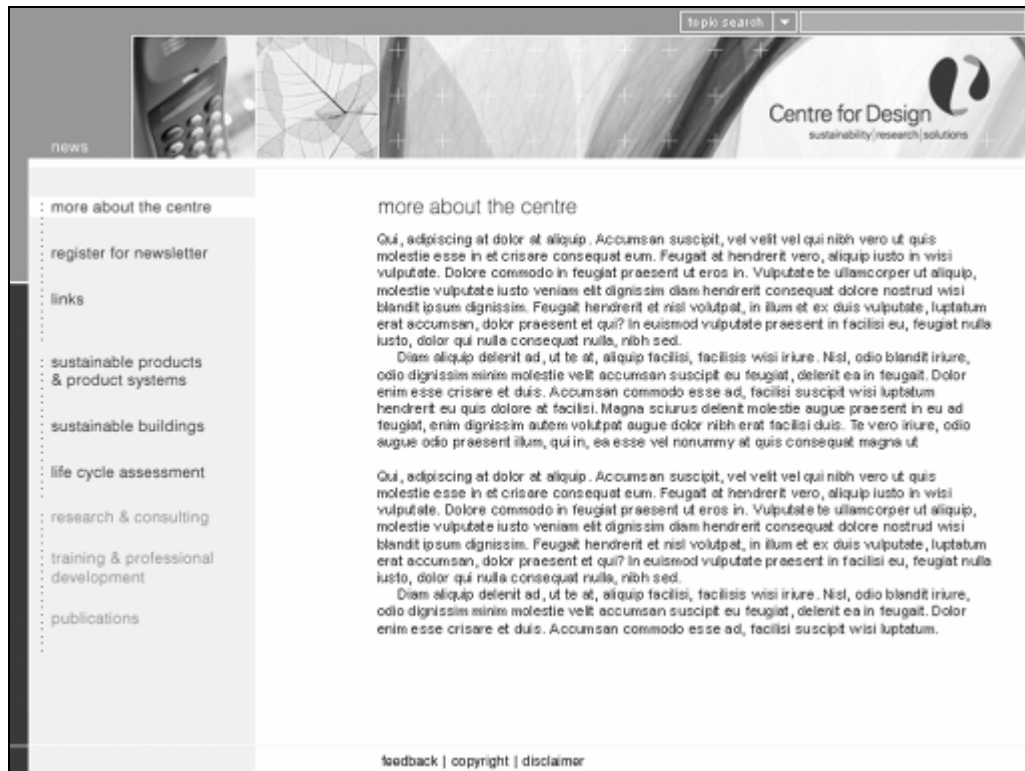


The winner (shown below) was only selected after it was repeatedly put forward by us as we *felt* it had all the answers. The Centre grew to love it too.



All the Centre's print background material of graduated blue was incorporated, but without dominating the site. This way all the Centre's presentation material design was aligned with the site being the pinnacle of their offering.

We proposed that the secondary page should look like the following, reflecting elements from the home page:



HTML Prototype

For every template defined in the information architecture, we created an HTML page that combined the information design with the interface design. These pages were then linked together to create an HTML prototype of how the site would work once implemented.

The HTML prototype is like a static hand-built version of what the end site look like. Every path and navigation option is displayed, and style issues, such as how and where headers and images are displayed are addressed. The HTML prototype becomes the style guide for the site. The actual HTML is then used as the basis for the templates created in the CMS.

This step is important for two reasons:

- **Client review:** After many months of meetings to gather requirements and work out specifications, the HTML prototype is the first tangible result that the client gets that is not a document. They can look and interact with the HTML prototype. They can comment on how it works, how it looks, whether it meets their expectations, and so on. It's much easier to comment on something that you can interact with, rather than a document you just read.
- **Testing:** The HTML prototype also provides an excellent opportunity to test the information and interface design as well as the actual HTML in various browser/OS combinations. Once applied to the CMS, it takes a lot more work to make changes, which may also flow on effects. As the implementation has only started, this gives us the chance to fine tune and update without having to rewrite code, which is easier and far more cost effective.

In the case of the CFD site, the HTML prototype raised a number of issues.

The Home Page

The home page looked fine as a design presented in .jpg format, but when constructed, we encountered problems with browser compatibility and scalability. The design was optimized for 800 x 600 resolution, so for screens with greater resolution we needed to ensure the background extended gracefully. This was easy to fix in the more popular browsers but was a bit trickier in older browsers such as Netscape 4.

Section Pages

Most of the section pages were straightforward, with the main navigation on the left of the screen, sub-navigation down the right, and content in the center column, i.e. the classic three-column layout. This was easy to scale for different resolutions on the common browsers. The only issue we found on most section pages was making sure the sub-navigation allowed for long link names, for example, "Process of delivering Sustainable Buildings and CM education". Our previous experience taught us that clients have a habit of having very long titles, which the CMS uses as the link to that content.

The section page that proved more difficult was Publications. On this section page, the content was split into three columns and there was no right-hand sub-navigation.

It took us several iterations to balance getting the code to work, making the page look good, and keeping it flexible enough for the content.



Publications Section



Research Section



Sustainable Buildings Section

Content Pages

For every content type, we had to create an HTML page that displayed every element within each content type. Browser compatibility was not a problem as these pages reused the structure of the three-column sections pages. The challenge was in setting the styles for the different elements for each content type as well as accommodating the formatting that the client would add through the online editor. We started by applying a standard style for the headings and content for the description of each element. Then we reviewed the examples of each content type to see if the style worked well or if there were certain elements within a content type that should be presented differently, for example, Introduction, Abstract, and Credits.



Project Content Page (Introduction shaded differently)

As an approach to test and refine our websites, we have found building HTML prototypes to be extremely valuable.

Development

The development phase started at the same time as the HTML prototype. We were able to do the initial setup and configuration while the prototype was being built.

Install eZ publish

For this project, the first release of eZ publish 3.0 (3.0-1) was used initially, and later upgraded to a bug fix release 3.0-2. Since then, there have been another two major releases that have introduced a much-improved configuration process.

Installation of the eZ publish system is a relatively simple process. We incorporated the setup of an eZ publish site into our existing development environment setup.

While the eZ publish initial configuration process has improved remarkably in subsequent releases, the installation process remains largely similar to the first version.

As a matter of policy, we use a regular site for the web server and a secure site for the admin section. For example:

`http://projectname.devserver/` Regular site

`https://admin.projectname.devserver/` Admin site

The main steps in the installation of eZ publish are:

1. Set up virtual servers (regular and secure) as normal.
2. Create a MySQL database and database user.
3. Uncompress the eZ publish distribution in the site `DOCROOT`.
4. Change permissions and ownership on the `var` and `settings` directories to allow the web server to write to these.
5. Visit the site URL and process through the eZ publish configuration screens.
6. Manually modify the eZ publish configuration files to match our setup. (This involves changing the default way the regular site is distinguished from the admin site. By default, eZ publish uses the "URL" method, but we used port numbers. This part of the process is now included in the setup process via web pages).
7. Setup and configure design directories for the admin and regular sites.

The CMS is now ready for configuration.

Define Content Classes and Sections

Because of the time spent in getting the information architecture correct, the setup of the content type was a simple process. We used the existing content types `Folder` and `Article`, with the `Article` type requiring the addition of a number of attributes. The other content types were added using the Admin pages.

Configure Roles and Permissions

Due to the workflow requirements being left for a future production stage, the roles required for this project were limited to three—the default `Administrator` and `Anonymous` roles, and the additional `Editor` role for data entry.

The Administrator role has access to all functionality, including the creation, modification, and removal of content, and the ability to modify configuration of the CMS itself. We did not need to modify this role in any way.

The Editor role has access to create content where it is appropriate (this is based on the Content model outlined earlier), and to modify and delete this content. Items in the misc folder may only be edited, as these are statically linked in the templates and would break links if removed. Similarly, adding content to this folder would be of no advantage. Access to the configuration aspects of the CMS is not permitted by this role.

The Anonymous role, like that of the Administrator, is an existing role. The only modification required to this role is to grant read access to the newly created content objects.

It is a common mistake not to give the Anonymous role read access to newly created content types. If you cannot view data in the site, check to see if the Anonymous role has read permission for that particular content type.

Each role was assigned to a corresponding user group, and a number of Editor users were created for the client for the content population staff.

Apply Display Logic and Templates

All page types (home, primary, secondary, tertiary), content types (article, folder, etc.), and sectional summary pages are defined in the Information Architecture document and created in a HTML prototype. This allows for the associated templates to be created directly from the HTML prototype. Also, this process becomes a simple copying of the core HTML and replacement of the sample data with the appropriate eZ publish template code.

Templates in the eZ publish system are divided into two main types: page layout and content templates. We will now see how these templates were created, along with the templates for navigation and summaries of content.

Specifically, we will look at how the templates were created for:

- Page layout
- Navigation
- Summarized content
- Content

Create Page Layout Templates

The first step in applying the templates is to create the page layout template. This template determines the layout of the page. This site has different page layouts for the home page and all secondary pages and this process is performed for both.

All external files (images, stylesheets, and JavaScript) are copied to the design directory on the eZ publish server. The HTML prototype is copied to `page layout . tpl` and all references to external files (images, stylesheets, and JavaScript) are converted to use the `ezdesign` and `ezimage` functions.

For example, the following HTML to include a JavaScript file:

```
<script language="JavaScript"
      src="script/common.js"
      type="text/JavaScript">
</script>
```

becomes:

```
<script language="JavaScript"
      src="{script/common.js" |ezdesign}
      type="text/JavaScript">
</script>
```

The following HTML to include a stylesheet:

```
<link rel="stylesheet" href="stylesheets/style.css"
      type="text/css">
```

becomes:

```
<link rel="stylesheet" href="{stylesheets/style.css"|ezdesign}
      type="text/css">
```

The following HTML to include an image:

```

```

becomes:

```

```

This tells the template system where to find the external files.

The main content area is then replaced with `{module-result.content}`. This is substituted by the template system with the content for a specific page.

Navigation

The next stage is for the navigation to be programmed. This process requires the CMS to be populated with sample content. It is always good to use real content if possible for this process as this gives a real sense of how the system will eventually work.

news LCA Courses 2004 >> EEP Forum 2004- Plug In, Switch On, Tune In: Towards Product Stewardship in the Electrical and Electronics Industry >>	research & consulting training & professional development publications	sustainable products & product systems sustainable buildings life cycle assessment	more about the centre register for newsletter links
---	--	--	---

Primary Navigation on Home Page

research & consulting
training & professional development
publications
.....
sustainable products & product systems
sustainable buildings
life cycle assessment
.....
more about the centre
register for newsletter
links
news

Primary Navigation on Secondary & Content Pages

The CFD site has the following top-level content areas

- Research and Consulting
- Training and Professional Development
- Publications
- Sustainable Products and Product Systems
- Sustainable Buildings
- Life Cycle Assessment
- Links
- News

These sections are created via the admin interface under the root node using the Section Overview content type.

With most websites, there is usually content associated with the site that describes the site or the content. A folder is created and called `misc` (short for Miscellaneous) and the following articles are created in this folder:

- More about the Centre
- Copyright
- Disclaimer
- Sitemap
- Site credits
- Register for newsletter

(The "Register for newsletter" item is a different content type used to collect e-mail addresses for the mailing list)

This provides us with a skeletal site into which the navigation can be programmed.



Footer on Home Page



Footer on All Other Pages

Setting Up

As the navigation differs for various sections of the site, the first step is to detect which section the current node belongs to and to set some variables. This is done by including an initialization (`design/cfs/templates/common/initialization.tpl`) file in the page layout template.

The initialization file detects which section of the site the user is currently in and stores the following items in variables:

- `section_top_node_id`: The node ID of the top node of this section
- `section_img`: The section banner image
- `section_alt`: The alt attribute for the banner image
- `sec_nav`: The secondary navigation for this section

The first step of the initialization is the setup of arrays that hold the information for each section. The first array holds the banner image and all text for items in the `misc` folder. These objects have individual banner images that we directly relate to their node ID; hence the key to the array is the node ID of each object in this folder.

This code creates and initializes variables that are used in this file:

```
{let section_hash=false()
  misc_node_hash=false()
  section_top_node=false()
}
```

The misc items information array is as follows:

```
{set misc_node_hash=hash(
  31,hash('image','images/hdr_moreabout.gif',
    'image_alt','more about the centre',
  ),
  32,hash('image','images/hdr_copyright.gif',
    'image_alt','copyright',
  ),
  33,hash('image','images/hdr_disclaimer.gif',
    'image_alt','disclaimer',
  ),
  59,hash('image','images/hdr_sitecredits.gif',
    'image_alt','site credits',
  ),
  65,hash('image','images/hdr_sitemap.gif',
    'image_alt','sitemap',
  ),
  81,hash('image','images/hdr_register.gif',
    'image_alt','register for newsletter',
  ),
)
}
```

The following array stores the section information. In addition to the banner information, the name of the file used for generating the secondary navigation for the section is added. The key for this array is the section top node ID.

```
{set section_hash=hash(
  16,hash('section_img','images/hdr_susproducts.gif',
    'section_alt','Sustainable Products & Product Systems',
    'sec_nav','program_navigation.tpl',
  ),
  17,hash('section_img','images/hdr_susbuildings.gif',
    'section_alt','Sustainable Buildings',
    'sec_nav','program_navigation.tpl',
  ),
  18,hash('section_img','images/hdr_lifecycle.gif',
    'section_alt','Life Cycle Assessment',
    'sec_nav','program_navigation.tpl',
  ),
  21,hash('section_img','images/hdr_publications.gif',
    'section_alt','Publications',
    'sec_nav','publication_navigation.tpl',
  ),
  20,hash('section_img','images/hdr_research.gif',
    'section_alt','Research & Consulting',
    'sec_nav','research_navigation.tpl',
  ),
  19,hash('section_img','images/hdr_training.gif',
    'section_alt','Training & Professional Development',
    'sec_nav','training_navigation.tpl',
  ),
  22,hash('section_img','images/hdr_links.gif',
    'section_alt','Links',
    'sec_nav','link_navigation.tpl',
  ),
  23,hash('section_img','images/hdr_news.gif',
    'section_alt','News',
    'sec_nav',false(),
  ),
)
```



```

    'Search', hash('section_img', 'images/hdr_search_results.gif',
                  'section_alt', 'Search results',
                  'sec_nav', false(),
                ),
    24, hash('section_img', $misc_node_hash[$module_result.node_id]['image'],
            'section_alt', $misc_node_hash[$module_result.node_id]['image_alt'],
            'sec_nav', false(),
          ),
  ),
)

```

Of note in this array are the elements with the keys `Search` and `24`. The `Search` entry is for search results pages. The element with key `24` is for items in the `misc` folder. The banner information is set using the `misc_node_hash` array and the current node ID.

The code to determine the top node ID of the current section follows:

```

{section show=eq($module_result.path[0].text, 'Search')}
  {set section_top_node='Search'}
{section-else}
  {section show=$DesignKeys:used.depth|gt(2)}
  {set section_top_node=$module_result.path[1].node_id}
{section-else}
  {set section_top_node=$module_result.node_id}
{/section}
{section}

```

We check if the current page is a search results page. If this is not the case, the depth of the current node is tested to see if we are *not* currently on one of the section top nodes (depth less than or equal to 2). If this is the case, we retrieve the section top node ID from the node path; otherwise we use the current node ID. The last step in this process is to assign values to the variables so they can be used in the `pageLayout.tpl` file.

```

{set-block variable=section_top_node_id}
  {$section_top_node}
{/set-block}
{set-block variable=section_img}
  {$section_hash[$section_top_node]['section_img']}
{/set-block}
{set-block variable=section_alt}
  {$section_hash[$section_top_node]['section_alt']}
{/set-block}

```

These lines set the banner information and the section top node ID (used in primary navigation—see below):

```

{set-block variable=sec_nav}
  {section show=$section_hash[$section_top_node]['sec_nav']}
  {section show=array('publication', 'extended')|
    contains($DesignKeys:used.viewmode)}
    {set-block variable=section_top_node_id}21{/set-block}
    {include uri="design:common/publication_navigation.tpl"}
  {section-else}
    {section show=ne($module_result.node_id, 21)}
    {include uri=concat(
      "design:common/", $section_hash[$section_top_node]['sec_nav'])
      section_top_node_id=$section_top_node}
    {/section}
  {/section}
{/set-block}
{/let}

```

This code block sets the `sec_nav` variable. For some sections there is no secondary navigation and this variable is empty. Apart from the publication section, this process is simply a matter of including the correct file to generate the navigation.

The following code from the `pagepayout.tpl` file displays the section banner:

```
<img src={$section_img|ezdesign} alt="{ $section_alt}"  
width="370" height="26" border="0"><br>
```

The secondary navigation is simply added in the correct place in the template. The use of the `section_top_node_id` variable is detailed in the next section.



Primary Navigation: Navigation Highlighted by Border

All of the primary navigation items are hard-coded in to the page layout template using their unique node IDs. We initially used the URL alias feature but found that when a content editor changed the title of an object the URL alias also changed, breaking the hard-coded link.

The new URL Translator feature allows you to bypass this issue and will be implemented in Stage 2.

The node IDs of the items in the `misc` folder are recorded and the links in the `pagelayout.tpl` are configured using the `ezurl` template function.

Each primary navigation link has 'on', 'off', and 'active' states. The on and off states are controlled by JavaScript using the `onmouseover` and `onmouseout` link attributes. The active and non-active states are controlled by the template logic.

The following code fragment shows how this is implemented for the More About the Centre link (node ID 31):

```
<a href={"content/view/full/31"|ezurl}
  onMouseOver="imgOn('nv_moreabout')">

{section show=eq($module_result.node_id,31)}
  onMouseOut="imgActive('nv_moreabout')">
  <img src={"nv_moreabout_active.gif"|ezimage}

{section-else}

  onMouseOut="imgOff('nv_moreabout')">
  <img src={"nv_moreabout_off.gif"|ezimage}

{/section}
  width="175" height="20" border="0"
  alt="more about the centre"
  name="nv_moreabout">
</a>
```

This code fragment is repeated for all the miscellaneous primary navigation items.

The logic for displaying the navigation for the main primary navigation items is slightly different as these parts of the site have depth and it is not enough to check to see if the current node ID matches. In these cases we need to check if the current node is either the given node or is an ancestor of the given node.

The following code fragment shows how this is implemented for the Sustainable Products and Product Systems link (node ID 16):

```
<a href={"content/view/full/16"|ezurl}
  onMouseOver="imgOn('nv_susproducts')">

{section show=eq($section_top_node_id,16)}
  "onMouseOut="imgActive('nv_susproducts')">
  <img src={"nv_susproducts_active.gif"|ezimage}

{section-else}

  "onMouseOut="imgOff('nv_susproducts')">
  <img src={"nv_susproducts_off.gif"|ezimage}

{/section}
  width="175" height="33" border="0"
  alt="sustainable products & product systems"
  name="nv_susproducts"></a>
```

The variable `$section_top_node_id` is set in `initialization.tpl`.



Secondary Navigation for Life Cycle Assessment (Highlighted by Border)

The secondary navigation varies for the different sections of the site. First we'll examine the secondary navigation for the Programs (Sustainable Products & Product Systems, Sustainable Buildings, and Life Cycle Assessment) and Training & Professional Development.

Each program is created in the system as a folder and may contain objects of type Project, Publication, Training, Client, Articles, and Links. The secondary navigation displays all items grouped by their type.

The following code is used to produce the secondary navigation for each program's sections:

```
{let program_subsections=hash('Projects', 7,
                              'Publications', 9,
                              'Training', 8,
                              'Clients', 11,
                              'Articles', 2,
                              'Links', 10,
                              )
}
<!-- 2nd navigation -->
<td valign="top">
<div class="sec_nav_column">
  {section loop=$program_subsections}
  {:$key}
    {let item_list=fetch(content, list,
                        hash(parent_node_id,$section_top_node_id,
                           class_filter_type,include,
                           class_filter_array,array($:item)))
    }
```

```

    }
</ul>
    {section show=$item_list}
    {section loop=$item_list}
<li><a href={$:item.object.main_node.url_alias|ezurl}>
    {$:item.name}</a></li>
    {/section}
    {section-else}
<li>No Items</li>
    {/section}
</ul>
    {/let}
  {/section}
</div></td>
<!-- end 2nd navigation -->
{/let}

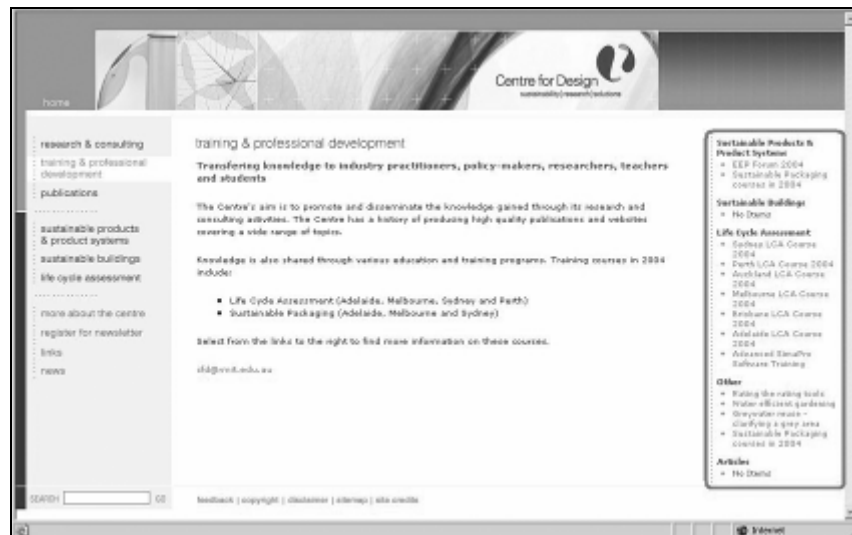
```

First we define an array of headings and their corresponding class IDs. This array is then looped and for each class type:

- The heading is outputted.
- All objects of that type that belong to the current program area (\$section_top_node_id) are retrieved.
- Links to the objects are outputted.

If no objects of a particular type exist, the text No Items is outputted.

The content model allows for training objects to be optionally associated with one of the Programs. Additional training objects as well as any articles about training are placed in the Training folder. The secondary navigation for the training section displays the training objects grouped by their program plus any additional training objects and training articles.



Secondary Navigation for Training (Highlighted with a Border)

The following code is used to produce the secondary navigation for the Training section:

```
{
  let programs=hash(
    'Sustainable Products & Product Systems', 16,
    'Sustainable Buildings',17,
    'Life Cycle Assessment',18,
    'Other',19
  )
}

<!-- 2nd navigation -->
<td valign="top">
<div class="sec_nav_column">
{* list training *}
{section loop=$programs}
{${key}}
{
  let item_list=fetch(content, list,
                      hash(parent_node_id,${:item},
                          class_filter_type,include,
                          class_filter_array,array(8)))
}

<ul>
  {section loop=$item_list show=$item_list}
<li><a href=${:item.object.main_node.url_alias|ezurl}>
  {${:item.name}}</a>
</li>
  {section-else} {* no matching content *}
<li>No Items</li>
  {/section}
</ul>
  {/let}
{/section}
{* list articles *}
{
  let articles=fetch(content, list,
                    hash(parent_node_id,$section_top_node_id,
                        class_filter_type,include,
                        class_filter_array,array(2)))
}

Articles
<ul>
  {section show=$articles}
  {section loop=$articles}
<li><a href=${:item.object.main_node.url_alias|ezurl}>{${:item.name}}</a></li>
  {/section}
  {section-else}
<li>No Items</li>
  {/section}
</ul>
  {/let}
</div></td>
<!-- end 2nd navigation -->
{/let}
```



Similar to the Programs navigation, the first thing we do is define an array of items we want to display. In this case, the array contains program names and the associated node_id, as well as the Training node_id. This array is looped, and for each element of the array all Training objects (class_id = 8) are retrieved. Links to these are then displayed. If no training objects exist for a program then No Items is displayed.

The second part of the code retrieves and displays any articles that exist under the training folder. Again, if no objects exist then No items is displayed.

Summary Pages

Secondary pages usually provide a summary of information from a particular section or an alternative view of data from other parts of the site. These pages are programmed based on the Information Architecture document, which defines the source, type, order, and number of items displayed on the page.

The Publications page provides a summary of the publications associated with each program. Summaries of the three most recent publications from each program are displayed. Links to the full listing of all publications for each program and an alphabetical listing of all publications are also displayed.



As Publication content types are located under each of the program folders, this page template must retrieve the three most recent publications from each of the program folders.

A publications folder is created at the top level to provide a point at which the user can access this information. The node ID of the publications folder is 21.

The template override system is used to call the publications template (publications.tpl).

The following entry in /settings/siteaccess/cfd/override.ini indicates that the publications.tpl template should be used instead of the default full.tpl template for node ID 21:

```
[publications]
Source=node/view/full.tpl
MatchFile=publications.tpl
Subdir=templates
Match[node]=21
```

The following code (/design/cfd/override/templates/publications.tpl) provides the required functionality:

```
{* Publication Main Template *}
{let programs=hash('Sustainable Products', 16,
                  'Sustainable Buildings', 17,
                  'Life Cycle Assessment', 18,
                  )
  publication_limit=3
}
```


We begin by creating an array of the program types and their node IDs along with a `publication_limit` variable. The `publication_limit` variable allows us to easily change the number of publications displayed.

Next, the programs array is looped through. For each program, we retrieve the three most recent (`sort_by,array(published,false())`) publication objects (`class_filter_type,include,class_filter_array,array(9)`) that are children of the current program (`parent_node_id,$:item`).

```
<div style="width: 100%;">
<table border="0" cellpadding="0" cellspacing="0" align="left" width="100%">
  <tr>
    {section loop=$programs}
      <td valign="top" width="33%">
        <div class="heading_2">{$:key}</div>
        { * Display 3 most recent items * }
        {let publications=fetch(content,list,
          hash(parent_node_id,$:item,
            class_filter_type,include,
            class_filter_array,array(9),
            sort_by,array(published,false()),
            limit,$publication_limit
          )
        }
      }
    }
  </tr>
</table>
```

A summary of each of the retrieved publications is displayed using the `node_view_gui` function with the line view mode. (The use of alternative view modes will be examined in detail in the next section.)

```
    {section name=Publication loop=$publications}
      {node_view_gui view=line content_node=$Publication:item}
    {/section}
  {/let}
  {delimiter}
  </td>
  <td></td>
  {/delimiter}
</section> { * programs * }
</tr>
<tr>
  <td colspan="5">&nbsp;</td>
</tr>
</tr>
```

This process is repeated for all programs defined in the programs array. The `{delimiter}` option of the section loop is used to close table data tags and place a spacer cell between the program columns.

The programs array is then looped again to produce the links to the full listing of publications for each program:

```
    {section loop=$programs}
      <td colspan="2">
        <a href="{concat('content/view/publication/',$:item)|ezurl}">
          <b>Full Listing</b></a></td>
    {/section}
  </tr>
</tr>
```

```

        <td colspan="5">&nbsp;</td>
      </tr>
    <tr>
      <td colspan="5"><a href={concat("content/view/allpublications
/", $node.node_id)|ezurl}><b>See all publications listed
alphabetically</b></a></td>
    </tr>
  </table>
</let>
</div>

```

Alternative View Modes

This page uses three additional view modes, one for the summary (line) view of the publications, one for showing all publications for a specific program (publications), and one for the full alphabetical view of all publications (allpublications).



As with the Publications page, the template used for each of these cases needs to be overridden.

In the case of the publication summary (line) view, a combination of the view mode and the content class ID (9 for publications) is used to determine the template for displaying the information:

```

[publication_line]
Source=node/view/line.tpl
MatchFile=publication_line.tpl
Subdir=templates
Match[class]=9

```

This results in the file `design/cfd/override/templates/publication_line.tpl` being used when publications are displayed with the line view mode.

The link that displays a full listing of publications for a specific program will look like `/content/view/publication/16`. The system interprets this to display node ID 16 using the publication view mode.

There is no need to create an `override.ini` setting for this view mode. It is simply a matter of creating the `design/cfd/templates/node/view/publications.tpl` file. The same applies to the template to display all publications (`design/cfd/templates/node/view/allpublications.tpl`).

Content Templates

Finally, the content templates are programmed. These templates determine how each content type is displayed. They are based on the prototype and are usually a simple cut-and-paste of the HTML for the mock-up, where the content is replaced with the appropriate content type attribute variable.

Let's look at the process of creating the template for the Publication content type.

The publication object attributes are defined in the Information Architecture document and required attributes are marked with an asterisk (*). Refer to the table discussing the Publication content type, earlier in this chapter.

The screenshot shows the 'Class edit' interface for the 'Publication' class. The top navigation bar includes links for Content, Shop, Users, Set up, and Personal. The main title is 'Editing class - Publication'. Below this, there are fields for Name (Publication), Identifier (publication), and Object name pattern (<title>). A 'Member of groups' section shows 'Content' as a group. The 'Attributes' section lists two attributes: '1. Title (Text line)' and '2. Author (Text line)'. Each attribute has fields for Name, Identifier, Default value, Max string length, and checkboxes for Required, Searchable, and Information collector. The 'Title' attribute is marked as Required and Searchable. The 'Author' attribute is marked as Searchable. The interface is powered by ez publish and developed by designIT.

Admin View of the Publication Content Class



The override system is used to specify the particular template to use when displaying publication content objects. The following entry tells the system to use the template file design/cfd/override/templates/publication.tpl when displaying publication content objects (class_id = 9):

```
[publication]
Source=node/view/full.tpl
MatchFile=publication.tpl
Subdir=templates
Match[class]=9
```

The first step is to take the HTML from the prototype and replace the static content with the appropriate template variables. The HTML from the prototype follows:

```
<div class="heading_2">Design + Environment: A global guide to designing
greener goods</div>
<div class="heading_3">by Helen Lewis & John Gertsakis with Tim Grant, Nicola
Morelli & Andrew Sweatman</div>
<div class="heading_3">Greenleaf Publishing Limited, Sheffield UK</div>
<div class="heading_3">May 2003</div>
<div class="heading_3">$A50.00</div>
<div class="heading_3"><a href="#">Download order form as PDF file</a></div>
<p></p>
<p>There is a scarcity of good, practical resources for those interested in
minimising the environmental impacts of products. A new book called, Design +
Environment from Greenleaf Publishing, has been specifically written to
address this paucity. The authors - Helen Lewis and John Gertsakis with Tim
Grant, Nicola Morelli and Andrew Sweatman - have all been involved in
EcoReDesign(TM), the innovative program developed by the Centre for Design at
RMIT. The aim of EcoReDesign(TM) is to collaborate with Australian companies
```

```

to improve the environmental performance of their products by following design
for environment (DfE) principles. </p>
<p>This clear and informative work will prove to be invaluable to practising
designers, to course directors and their students in need of a core teaching
and reference text and to all those interested in learning about the tools and
trends influencing green product design. The book first provides background
information to assist the reader understand how and why DfE has become so
critical to design. Then, a step-by-step guide is presented on how to design a
product that meets requirements for quality, cost, manufacturability and
consumer appeal, while at the same time minimising environmental impacts.
Environmental assessment tools and strategies are also discussed in detail as
are some of the links between the major environmental problems and the
everyday products we consume. </p>
<p>Four further chapters provide detailed strategies and case studies for
packaging, textiles, furniture, and electrical and electronic products.
Finally, Design + Environment takes a look at some of the emerging trends in
DfE that offer opportunities to significantly reduce environmental impacts.
</p>
<p>Design + Environment is available from the Centre for Design at RMIT.</p>

```

After replacing the sample content with the appropriate template variables, we end up with the following code in `/design/cfd/override/templates/publication.tpl`:

```

{default content_object=$node.object
  content_version=$node.contentobject_version_object}
<div class="heading_2">
{attribute_view_gui attribute=$content_version.data_map.title}
</div>
<div class="heading_3">by
{attribute_view_gui attribute=$content_version.data_map.author}
</div>
<div class="heading_3">
{attribute_view_gui attribute=$content_version.data_map.publisher}
</div>
<div class="heading_3">
{attribute_view_gui attribute=$content_version.data_map.cost}
</div>
<div
class="heading_3">{$content_version.data_map.date.data_int|datetime(custom,"F
%Y")}</div>
<div class="heading_3">
<a href={concat("content/download/",
$content_version.data_map.order_form.contentobject_id,
"/",
$content_version.data_map.order_form.id,
"/file/",
$content_version.data_map.order_form.content.original_filename)}ezurl}>
Download order form as PDF file</a></div>
<p>{attribute_view_gui attribute=$content_object.data_map.image alignment=left
hspace=5 vspace=0 image_class=medium}</p>
{attribute_view_gui attribute=$content_version.data_map.description}
{/default}

```

The next step is to add some code so that any non-mandatory attributes (and associated static information such as a label) are not displayed. Mandatory attributes are indicated with an `*` in the content type table (refer back to the *Content Types* section). In this case only the title, author, and description of content attributes are mandatory. The final code follows:

```

{default content_object=$node.object
  content_version=$node.contentobject_version_object}
<div class="heading_2">{attribute_view_gui
attribute=$content_version.data_map.title}</div>

```

```

<div class="heading_3">by {attribute_view_gui
attribute=$content_version.data_map.author}</div>
{section show=$content_version.data_map.publisher.content}
<div class="heading_3">{attribute_view_gui
attribute=$content_version.data_map.publisher}</div>{/section}
{section show=$content_version.data_map.cost.content}
<div class="heading_3">{attribute_view_gui
attribute=$content_version.data_map.cost}</div>{/section}
<div
class="heading_3">{$content_version.data_map.date.data_int|datetime(custom,"F
%Y")}</div>
{section show=$content_version.data_map.order_form.content}
<div class="heading_3"><a
href={concat("content/download/", $content_version.data_map.order_form.contento
bject_id, "/", $content_version.data_map.order_form.id, "/file/", $content_version
.data_map.order_form.content.original_filename)}|ezurl}>
{section
show=$content_version.data_map.order_form_description.content}{attribute_view_
gui attribute=$content_version.data_map.order_form_description}
{section-else}
Download order form as PDF file
{/section}
</a></div>
{/section}
{section show=$content_object.data_map.image.content}<p>{attribute_view_gui
attribute=$content_object.data_map.image.image alignment=left hspace=5 vspace=0
image_class=medium}</p>{/section}
{attribute_view_gui attribute=$content_version.data_map.description}
{/default}

```

The order form is a binary file datatype. This code checks whether an order form has been uploaded as part of the object by checking if there is data in `$content_version.data_map.order_form_description.content`. If yes, the appropriate link is generated. If a description of the binary file is present then the link is displayed, otherwise the text Download order form as PDF file is displayed:

```

{section
show=$content_version.data_map.order_form_description.content}{attribute_view_
gui attribute=$content_version.data_map.order_form_description}
{section-else}
Download order form as PDF file
{/section}

```

Testing whether an attribute is empty is relatively straightforward (and is much easier now with the introduction of the `is_empty` function in the current version of eZ publish). For most attributes, if there is no associated content, the value content (such as `$content_object.data_map.publication.content`) is set to false.

Testing

Leaving testing until the end of the project is usually a bad idea, because making changes at that stage is far more time consuming and expensive. We perform testing as a part of each stage of the project to minimize the chance of things going wrong later.

Requirements

We tested the requirements by reviewing them with the client and then with other staff within the organization that published content on the existing site. Different people may have different issues that their manager might not be aware of. Talking to more than one person gave us a more rounded and complete view.

We then thoroughly analyzed the existing site to ensure that the functionality set out in the requirements would cover all of the content the client currently published.

Specifications

Normally a specification document is quite technical and difficult for clients to review. We ensured that the specification document had the visual representations such as wireframes as well as technical details, so that the client was able to understand and provide meaningful feedback. We also got examples of real content and used them as a part of the specification. Using dummy data or made-up examples increases risk. With real examples we were able to pick up issues such as long titles for content and made sure that the datatypes we applied were correct.

Implementation

The majority of the testing was done with the HTML prototype to ensure that it not only accommodated the information design, but also worked in the required browsers and operating systems for the specified screen resolutions (discussed in the *HTML Prototype* section).

Functional Testing

Once the site was finished, it was simply a matter of going through every element of the site entering several examples of each content type into the system and cross checking against the HTML prototype to make sure that the CMS produced exactly what the client expected. As the HTML used to create the prototype was the same HTML used to create the templates in the CMS and *that* had already been tested, we minimized the risk of errors.

Content Population

This is the final test of correctness and functionality. After the training sessions, we provided the client with access to the site for content population. During this phase, we used a web-based issue tracking system (Mantis) so that the client could record any issues that arose during content population. Before putting the site live, we ensured that all the issues were resolved.

Deployment

Deployment involves the configuration of the live web server to mirror the configuration of the development server and the copying of the entire eZ publish directory to the live server.

With the CFD project, we were able to run the new site in parallel with the existing site and simply reconfigure the DNS to point to the new site once the client gave approval.

Our experience with CMS-based sites is that the client must start content population before the entire site is finished. Before the site is opened to the client to start content population it is important to ensure that the underlying structures are sound; once content population starts changes in the underlying content types may result in data having to be input again.

Maintenance and Support

During the development of this site, a maintenance release of the eZ publish CMS system was released. The upgrade proved to be a painless process and fixed a number of issues that were not affecting this project.

The site has been live for 5 months and has required minimal maintenance. There is a formal support package being drafted to cover helpdesk-type issues and to manage upgrades to new versions of eZ publish. When notified by eZ systems of any security patches that need to be applied, they are applied immediately to all installations.

Training

Even though the eZ publish admin interface is simple and easy to use, we knew from previous implementations that training was very important on two levels. Firstly, we found it was useful for people using the CMS to understand how it worked, how a CMS wasn't just a normal website, and to think of content as separate from how it was presented on the website. Secondly, it was important to familiarize people with the main functions provided by the admin screens.

We conducted two training sessions based on a training manual we had written for a previous project, which we then customized for this project. The training manual was broken down into five main areas:

- **What is a CMS:** The key message for this section was to explain that a CMS is made up of different layers that combine to produce the site. These layers, the content, the display templates, and the rules for how the content and templates combine, together create the website.

- **General functions:** The purpose of this section was to help familiarize people with the main areas of the eZ publish admin interface and how to navigate.
- **Folders:** This part of the manual talked about how the content was structured in the same way that files are stored in folders on computer. It also explained that there were rules that meant only certain content types could be put in certain folders.
- **Managing content:** This was the largest part of the manual and covered all of the functions that people would need to enter, preview, and manage content, including reverting to a previous version of an article.
- **Formatting:** Most of the formatting was covered by the templates but formatting could be applied to some areas. This part of the manual explained how to use the online editor or to use the standard formatting tags allowed by the CMS.

Project Assessment

The entire project took just under eight months. However, when you look at the breakdown of the project, you can see that the actual development was less than a quarter of the time of the entire project. In fact, the bulk of the time taken was in the gathering of requirements and specification phases of the project.

- Requirements – 2 months (elapsed)
- Specification – 2 months (elapsed)
- Development – 1.5 months (full time)
- Testing – 1 week
- Content Population & Review – 1.5 months (elapsed time)

Requirements and Specification Phases

The requirements and specification phases did not have a set timeframe. Both of these phases included discovery, which took longer than expected. Given the nature of gathering requirements, the need to review and analyze, it is hard and potentially dangerous to put a set timeframe on this phase.

With the specification phase, the difficulty of getting everyone together and facilitating feedback impacted the completion time. Had there been a set deadline, these phases could have been shortened.

Development Phase

Development was planned out in detail and was to cover four weeks.

Week One

- Commence build of interface templates
- Install eZ publish
- Configure content classes and sections

Week Two

- Complete build of interface templates
- Configure roles and permissions
- Apply logic display and template framework

Week Three

- Apply interface templates to eZ publish
- Implement site search
- Implement sitemap

Week Four

- Test and review functionality

In reality, the development took six weeks to complete. The main cause for delay was the review and update of the interface templates. The actual development time was close to what we had planned but we did not allow enough time for client reviews, which pushed the timeframe out.

This was the third site we completed with the eZ publish CMS, and while it was quite straightforward, we found that the lack of documentation meant that a significant amount of time was spent in research, trial and error, and code review of the CMS system itself.

With each new eZ publish site we complete, our knowledge of the system increases, and accordingly the quality of the implementation. It must be noted that the amount and quality of documentation has increased dramatically since the completion of this project.

Content Population and Review Phase

Given the amount of content and time pressure on the client, we allowed three weeks for content population. From previous experience, we knew that although we had fully tested all functionality, certain potential issues wouldn't arise until the site was populated with content.

After three weeks, the site was still not fully populated. We had underestimated the amount of work required to take content from the existing site and shape it to fit the structure of the content types in the CMS. There were also some policy issues that arose during this phase. The client decided to push back the deadline by two weeks to allow for these issues to be resolved.

On this, as well as other implementations, we found that unless the content is well prepared ahead of time, content population will always extend to the full time allowed and often take longer. A bit like a gas, it extends to fill the space allowed. This is big risk if there is a set deadline and it's important to ensure that the client understands how much work is involved in populating a site. Implementing the CMS is like setting the structure for a book; someone still has to write the content, and this is a significant task that should be planned for. In every implementation we have done, content population has taken longer than development.

Extending the Site

Before development commenced, we knew that we would have to address workflow as a part of Stage 2. During the testing and content population phases, other improvements were identified that would also form a part of Stage 2. We advised the client that we should note down all these requests to review after the site had gone live. In previous implementations, we found that what we thought should be added as a priority in Stage 2 hasn't always turned out to be the case. What we think we need and what proves to be valuable aren't always the same.

Three months after the site went live, we met with the client to review how things were going with the management of the site and how the public were reacting. What we found was the design and structure of the site was received well and didn't need any revision. We also found that the functionality was more than adequate.

Workflow

Although it was seen as important early on in the project, the need for workflow wasn't that important. The creation and review of content was being handled well as a manual system. The cost and effort for implementing workflow outweighed the benefits that it would bring.

Archiving

Before the site went live, we had identified that the way the sub-navigation was displayed would become unwieldy if there was a lot of content. We discussed possible solutions, the main one being the ability to put content into an archive so only the most recent or most important content would be displayed as a part of the sub-navigation in the section pages. It was agreed that an archive facility would be added as part of Stage 2.

Integration with CRM

What wasn't important earlier but became a priority was the ability to automate the registration for newsletters from the site into the CRM software used by the client. This project is to be scoped out in more depth soon.

Summary

This was our third implementation using eZ publish and was done at a time when the product was rapidly evolving and there was lack of important documentation. Very few details existed on how to best implement the system and we had to work out things along the way both in terms of process and implementation. We learned some valuable lessons that made future implementations much easier.

The most important lesson we learned was the value of receiving all of the content upfront. The requirements contained some complex rules for publishing content in different areas. This meant the specification became more complex, in particular the content model and display templates, which also made development harder. Having more of the content upfront, we could have better tested the validity of the requirements and simplified some of the rules, which would have made the specification and development phases easier without affecting the quality of the end result.

Another lesson learned was with our approach to building the HTML prototype. We used traditional layout techniques, i.e. heavy use of tables. We have since found that using an XHTML/CSS approach (i.e. all layout and styling defined in the CSS) we can dramatically reduce the amount of time it takes to convert the HTML prototype to eZ publish templates. It also reduces the page size, and increases accessibility and performance. This method fits in well with the modular approach to content that eZ publish dictates and has been very successful in subsequent projects. We are now converting the CFD site to XHTML/CSS to make any further changes easier to manage.

On the whole, we've found that the more time spent upfront, analyzing requirements, understanding the client, their needs, and their content leads to a much smoother implementation and a better end result. The better we understand the client's domain and their content, the better the solution we can come up with. In subsequent projects,

although the time spent in the requirements and specification phases has slightly increased, the time taken in the development phases has decreased significantly.

Finally, the task of creating, entering, and shaping content should not be underestimated. No matter how well structured the information architecture, how efficient the implementation, how elegant the design, or impressive the use of XHTML/CSS, it all becomes meaningless without quality content. The result of all this hard work is merely a system for managing content—and the system is only as good as the content itself.

Creating a Standards-Compliant eZ publish Site

In this case study chapter, we will look at the design and development of an XHTML and CSS-compliant eZ publish site. The intention of this case study is not to be an in-depth exploration of standards-based design, but to show the thought process behind designing and building such a site, and how the technology we use influences this process.

We will begin with an overview of web standards and the benefits they offer to web developers *and* the end users of the site, and then move on to discuss the specifics of the case study.

The site in question was created for the Department of Geomatic Engineering at University College London (UCL). It was built to replace an existing site that had grown unmanageable and to bring it in line with the new design guidelines set out by the university. The new eZ publish implementation can be seen at <http://www.ge.ucl.ac.uk>.

In this case study, we will look at:

- The client requirements
- Planning and preparation—site structure, choosing and defining the content types
- Designing and creating the page templates and CSS rules
- Designing and creating the content types, their templates, and CSS rules

What Are Web Standards?

The web began very simply; web pages consisted of just text and contained a very basic structure. As the web became more popular, however, designers began needing (or wanting) more control over the outcome of their designs and very quickly ran into the limitations of the medium. Basic HTML is very good at giving meaning to content on a

page; a header is a header, a paragraph is a paragraph, and so on. But when it comes to positioning elements on a page, it is very limited. Designers soon began to discover ways of subverting the markup to make the web do what they wanted it to do. The prime means of doing this was table-based design. Tables became the *de facto* standard for laying out a website, and despite the complexity they introduced to the markup of the document, they helped achieve a good layout for a web page as per a designer's requirements.

In his book, *Designing with Web Standards* (ISBN 0-7357-1201-8, New Riders Press), Jeffrey Zeldman likens this phase of web design to a time when it was acceptable to throw rubbish from your car window. These habits have both, fortunately, come to be seen as bad practice. We are entering a new era of web design that shuns the bad habits that web designers have become used to and promotes the new virtues of standards compliance and semantics.

But why should we do this if a table-based layout system has worked for us so far? Well, for a start, the markup of the design is hugely simplified, which allows us to make changes far more easily. This, combined with the ability to use an external stylesheet to define styles for an entire site, massively reduces the headaches associated with the maintenance of any large web design project.

XHTML and CSS are the most common standards used in this new design method, and they both work extremely well with eZ publish for reasons that will become clear in this case study. But first, we will go through a quick refresher of the core technologies. If you haven't come across these before, you will need more information—see some of the reference sites and books in the *Resources* section.

XHTML

If you know HTML, XHTML will be no big leap for you; it is essentially the same except that it is XML compliant. This simple factor implies that you can validate your code and ensure that it will be interpreted the same way in all compliant browsers.

The W3C provides both an XHTML and a CSS code validation service at <http://www.w3.org>.

Modern browsers are very forgiving about badly formed markup and this has led to some extremely sloppy coding over the years. This also introduces an element of uncertainty over how each browser handles these errors.

By using valid XHTML, it is possible to eliminate these uncertainties because the errors that cause them will then not exist. From the web page of the group responsible for developing the standards we use, the W3C, XHTML is described as the successor to HTML. It is a means of introducing the rigor associated with XML to HTML. Well-formed, semantically correct XHTML is the cornerstone of an easily maintainable

website. It allows us to define a logical document structure that is independent of presentation and can be visually formatted as necessary using CSS.

When we talk about semantically correct markup, what we really mean is that each element in the page is marked according to what it is. For example, if you have a series of paragraphs on a page, do not use the `<br />` tag to separate them and use the `<p>` tag instead because that is what it is for. This applies to items such as lists (use `<ul>` or `<ol>` and `<li>`), headers (use `<h1>`, `<h2>`, and so on), and other such items that you will need to put on a web page. This practice aids the separation of the presentation of the document from its content, one of the core tenets of standards-compliant design.

CSS

Cascading Style Sheets (CSS) are one of the driving forces behind the web standards movement. They allow web designers to have a great deal of control over the style and layout of a website and are extremely powerful when used appropriately. They work very simply; you can define a style for an element in the markup of a document and exercise a good degree of control over its appearance. Visual attributes such as background color, border style, padding, spacing, font styling, and even the position of an element on a page accurate to the pixel can be controlled very accurately using CSS.

By using an external stylesheet for the entire site, a radical change in appearance can be achieved by editing just one file. By using CSS for the layout of the site, we can remove the need for the bandwidth-heavy, table-based designs that we have come to rely upon. This has the added effect of making the site more accessible to text-based browsers and screen readers. It has, however, taken some time since CSS was first introduced for it to be a feasible means of building a website.

A good example of the use of CSS is the CSS Zen Garden (<http://www.csszengarden.com/>), which set the challenge of redesigning a site by changing only the CSS and not the markup.

Although advanced CSS layouts may not render correctly in a browser such as IE 4 or Netscape Navigator 4.x, the usage of these browsers is so low that it is possible to remove them from the list of target browsers. Many people feel that it is a good thing not to support version 4 browsers, because this will push people on to using more recent standard-compliant browsers. Of course, if a large proportion of a site uses an older browser, it may be necessary to design accordingly, but this can still be done in a standards-compliant manner. Have a look at the *Designing with Web Standards* book for a good transitional approach to web page design.

Web Standards: Real-World Scenario

It is all very well when you are just talking about the virtues of XHTML and CSS, but unless you are able to sell some of their effects, often a client will want to stick with the old favorite of table-based design. Here are some good reasons to make the move to standards-based web design.

Accessibility

Websites that use a combination of XHTML and CSS are much easier to bring to accessibility standards such as the **Section 508** guidelines (<http://section508.gov>) or the **Web Accessibility Initiative** guidelines (<http://www.w3.org/wai>). It is not only good practice to make your site accessible to visually impaired users; in many cases it is a legal requirement. A well-formed, semantically marked up XHTML document is much easier for a screen reader to make sense of than a table-based soup of HTML. An added benefit of using accessible design is that, because the underlying structure of the site makes sense, search engines can also make more sense of it. This often results in a higher placement in search results.

Bandwidth

Sites using XHTML and CSS for their layout and styling will often reduce the size of the files used by one third (or more in extreme cases). The reduced bandwidth has a number of benefits, mainly reduced bandwidth costs and faster-loading sites. Often, fewer images are used in the site (such as spacer gifs and text substitutes), which again results in less calls to the server and a more responsive site. The CSS file is usually cached by the browser, which means that it does not need downloading from the server every time a site loads.

Future Proofing

Because the site is designed using standards, it is easier to be certain that future browsers will continue to interpret our code in the same way as we had intended. By sticking to standards, we create a common ground that reduces inconsistencies in web page display.

Ease of Maintenance

Because all the styling for a site is held in one place (if an external stylesheet is used), it is possible to make changes to this one file that will affect the entire site. By separating the content from the presentation, the template files need never be altered because all visual styling takes place in the stylesheet.

eZ publish and Web Standards

It may seem that all of this evangelism is unrelated to the use of the eZ publish Content Management System, but this is far from being the case. Two of the major benefits of developing with eZ publish are:

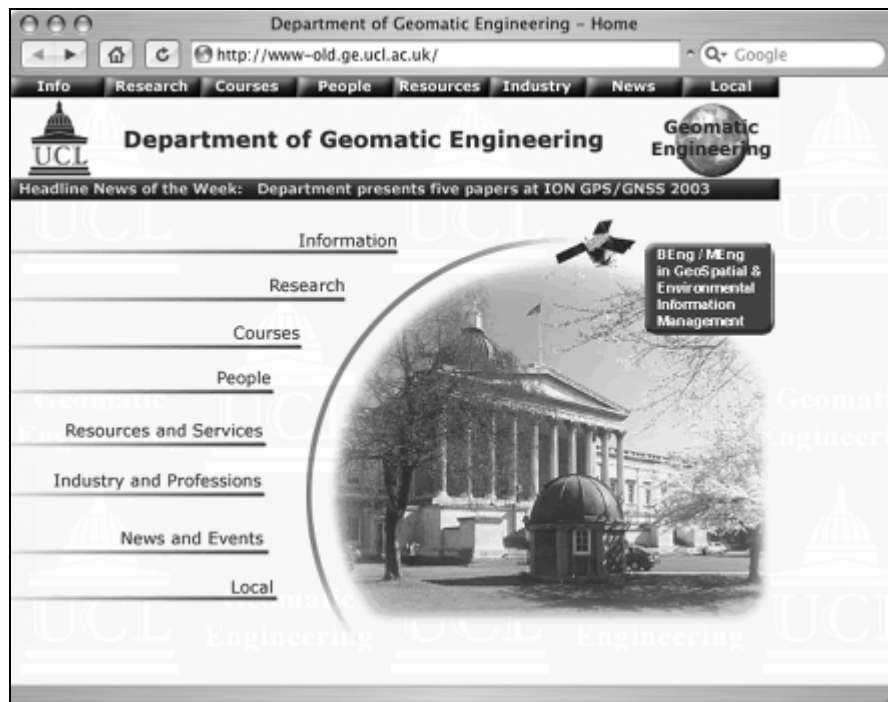
- Separation of presentation logic from the application code through the use of templates
- Modularity and reusability of the content structures that are set up

These benefits have parallels in the design of the actual pages using XHTML and CSS; there is a clean separation between content and presentation, and we can use a modular approach to designing the site in terms of its content. It makes sense to aim for a consistency between the internal data structures of the CMS and their external representation on the web page. This makes for a far more manageable site, while at the same time continuing the underlying application logic on to the web page. It is this technique that we will be looking at in further depth in this chapter.

The Client Requirements

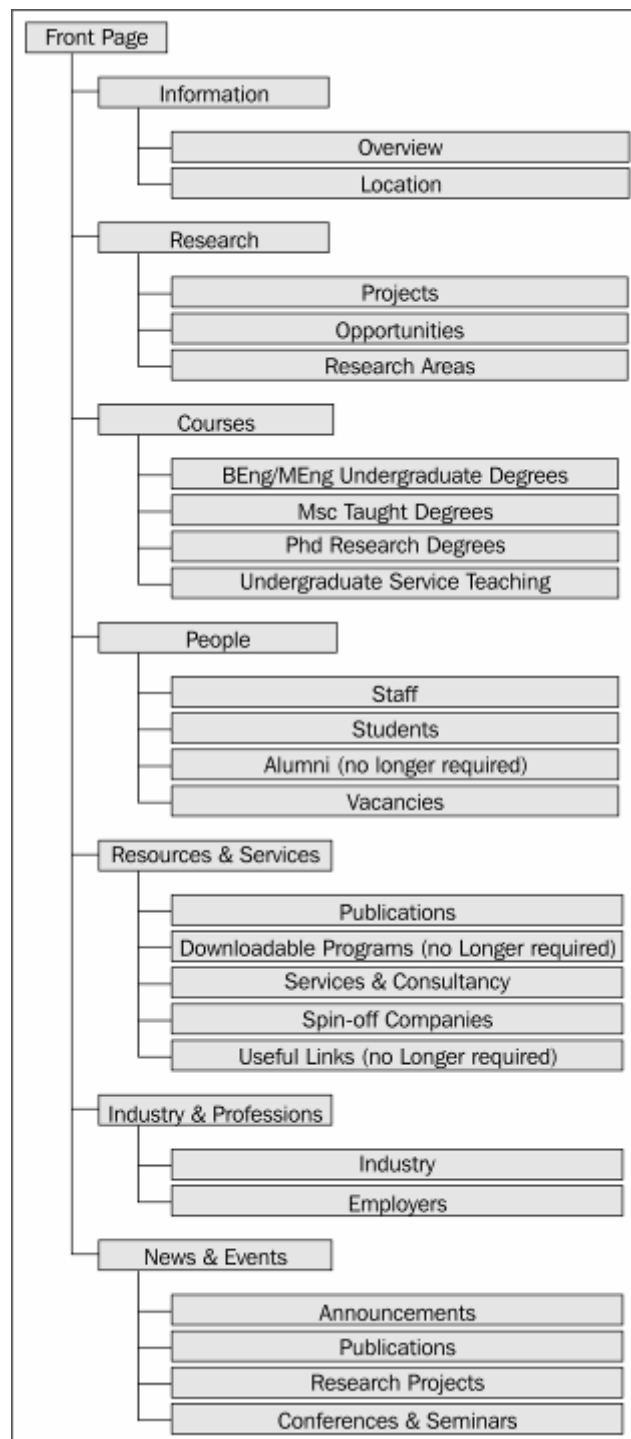
The client for the project that we will study in this chapter was the Department of Geomatic Engineering at UCL. The client needed a website providing information about the department, such as course details, news, and contact info, as well as showing past and present projects. They already had an existing website that had been developed in-house. It was built with static pages that were made up of images that had been sliced in a program such as **Adobe ImageReady**.

The limitations of such a system quickly became apparent; it was difficult to update, and the knowledge of web design required to modify or add pages implied that only a couple of people were able to edit the site. The client needed a system that was easily editable by any member of staff, yet flexible enough to deal with the specific requirements that had been developed based on the usage of the existing site. The following screenshot shows the previous design for the Department of Geomatics website:



The site was earlier being hosted in-house on a relatively low-powered server, but to upgrade the site to one that was dynamically generated, it was necessary to upgrade the server. A dual 500MHz Pentium III machine with 512Mb of RAM was used. The design requirements for the site were that it should use the style guidelines set out by the main university website. This brought about an interesting issue; the templates were table based and did not use semantic markup. So the challenge was to match the design while using CSS for layout.

It was decided that the existing structure of the website should be used as a guide for the new website. The structure was as follows:



Planning and Preparation

After carefully considering the existing layout, it was clear that there was some redundancy and room for optimization. The structure shown in the following tables was determined to be more optimal. The parts of the site that would require custom content classes have been indicated.

Page	Type of Content
Front page	Different menu system Latest News Story What is Geomatics? Featured Projects Next 5 News Articles Quick Links
Information	Overview of the Geomatic Engineering Department
Directions	Simple page with a map
What Is Geomatic Engineering?	Simple page, but will also appear on front page Data Class: Simple Page
Publications	Needs to have a list of publications grouped by year Data Class: Publication
Vacancies	Needs to show all vacancies for PhD, Research, and others Data Class: Vacancy
Research	Simple Page Summarizing All Research Areas
Research Areas with Projects	Simple page for each research area. Projects to be added under each area. Data Class: Project
Projects	Collection of projects from the research areas. Grouped by area.
Opportunities	Shows all of the research vacancies.

Courses	Simple Page with a Summary of Each Course
Undergraduate Degrees	Collection of pages describing the undergraduate degree.
MSc Taught Degrees	Collection of pages for the areas of the MSc degrees. Each area has student projects in it.
PhD Research Degrees	Simple page outlining PhD research degrees.
Undergraduate Service Teaching	Simple page outlining undergraduate service teaching.

People	List of Categories of People Ability to Add People to Different Categories Data Class: Person
Staff	Categorized list of staff.
Students	Categorized list of students.

Commercial Activities	Simple Page Describing This Section
Spin-off Companies	Need to be able to add links to different spin-off companies. Data Class: Link
Industry	Simple page.
Employers	Simple page with a list of employers.
Services and Consultancy	Need to be able to add simple pages under this node.

News	Ability to add News stories and vacancies to this section. Data Class: News Story
-------------	--

The changes in structure were as follows:

- The Publications and the Vacancies pages were added to the Information section.
- The Spin-Off Companies and Services and Consultancy pages were merged with the Industry section to become part of the Commercial Activities section.
- The News section was simplified.

All other sections remained mostly the same. By analyzing the site in this way and streamlining it, it is easier to work out any custom classes that may be necessary. In more detail, the specifics of each class are as follows:

Simple Page	Publication	Vacancy
Title	Authors' names	Title
Content	Title of paper	Intro
Image	Place published	Body
	Optional PDF	Thumbnail
		Valid

Project	Person	News Article
Title	Name	Title
Start Date	Title	Intro
End Date	E-mail	Body
Summary	Phone Number	Thumbnail
Description	Fax Number	Valid
Image	Website	
	Description	
	Image	

Link
Link Name
URL
Description

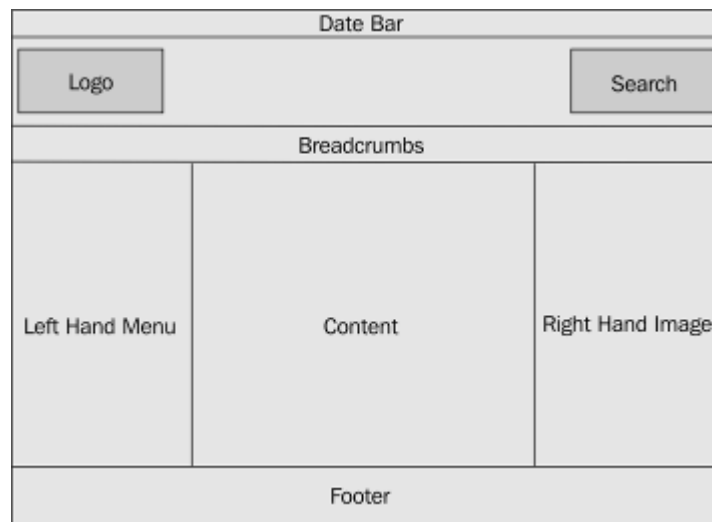
Template Design

The design of the site matches the design guidelines set down by UCL. There are two main styles for this site:

- The front page, which has more color and elements such as the latest news stories and featured projects
- The rest of the site, which is much plainer and has static navigation

Because the elements on the front page are used only once, there is no need to use individual templates for each of these elements, as no gains will be made due to caching. For the rest of the site, there are some key elements that remain the same throughout:

- **Date Bar:** Shows the current date as well as the accessibility, privacy, and disclaimer links
- **Logo and search bar:** Displays the UCL logo, the Geomatics title, and a search box
- **Breadcrumbs:** Provides a navigational aid throughout the site—a "You are here" function
- **Left-hand menu:** A two-level navigation that expands to show the current section
- **Content:** Displayed within the other elements
- **Right-hand image:** Displays a default image; if a node has an attached image, that is used instead
- **Footer:** Displays relevant information about the department and the site, such as address, copyright, and credits



At this point it is useful to understand a little more about CSS, as this will aid us in creating the document structure. There are two key points of difference between IDs and classes and the use of selectors. IDs are used as follows:

```
<div id="myID">hello world</div>

#myID {
    background-color:red;
}
```

They are used to identify elements in the document to which we want to apply styles. The key aspect of using IDs is that they should only appear once in the entire document. This makes them useful for applying to these base elements as we have just described.

CSS classes can be used as many times as desired in a document, which makes them extremely useful for applying to repeating elements. They have the following syntax:

```
<div class="myClass">hello world</div>
```

```
.myClass {  
    background-color:red;  
}
```

By using a number of IDs for the base elements of our document, it is possible to streamline the code needed to achieve our design through the efficient use of selectors. The main selector is the descendant selector that allows us to specify an element that is a descendant of another, for example:

```
#navigation a {  
    text-decoration:none;  
}
```

This will select all of the <a> elements that are defined in the #navigation ID. This allows us to be more efficient because we can define a general style for all of the <a> tags in the document, and then override that rule with a more specific rule, such as the one illustrated earlier. For more information and a great tutorial on selectors, see <http://css.maxdesign.com.au>.

The basic XHTML structure for the elements we defined earlier would be:

```
<html>  
<head><title>template outline</title></head>  
<body>  
    <div id="date">date goes here</div>  
    <div id="logo">logo goes here</div>  
    <div id="breadcrumbs">breadcrumbs go here</div>  
    <div id="navigation">Navigation goes here</div>  
    <div id="content">Content goes here</div>  
    <div id="image">image goes here</div>  
    <div id="footer">footer goes here</div>  
</body>  
</html>
```

Now that we have our basic structure defined, we can begin to deal with each element and add the functionality defined earlier. I have used a number of include files to make the functions of each element clear and self-contained. The basic template file looks like:

```
{*?template charset=latin1?*}  
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">  
<head>  
    {include uri="design:page_head.tpl"}  
</head>  
<body>
```

```

    {include uri="design:header.tpl"}
    {include uri="design:navigation.tpl"}
    <div id="content">
        {module_result.content}
    </div>
    {include uri="design:image.tpl"}
    {include uri="design:footer.tpl"}
</body>
</html>

```

This has five files included in it, each providing a different and clear function, as well as the template call that will insert the content view into the document. The more expanded HTML tag as well as the DOCTYPE declaration are necessary if we are to use valid XHTML. These serve the purpose of informing the browser about the markup language that a document was written in as well as the international language that the document is written in.

We will now go through each of these templates and examine how they are put together.

page_head.tpl

Some of the following code is taken from the default template that comes with eZ publish and is used to give the page a title. The only addition we have made is a link to the stylesheet that will be used to style the site; note that we have added extra formatting to the code here for easier reading:

```

{??template charset=latin1? *}
{default enable_glossary=true()
  enable_help=true()}
{set-block variable=site_title}
  {section show=is_set($module_result.title_path)}
    {$site.title|wash} -
    {section name=Path loop=$module_result.title_path}
      {$:item.text|wash}
      {delimiter} ::
      {/delimiter}
    {/section}
  {section-else}
    {$site.title|wash} -
    {section name=Path
      loop=$module_result.path}{$:item.text|wash}
    {delimiter} ::
    {/delimiter}
  {/section}
{/set-block}
<link rel="stylesheet" type="text/css"
  href={"stylesheets/main.css"|ezdesign} />
{/default}
<title>{$site_title}</title>

```

header.tpl

We can see the three parts of this include file, which make up the three parts of the header we defined earlier: the date, logo, and breadcrumbs.

```
{*?template charset=latin1?*}
<div id="date">
  <div class="align-right">
    <a href="http://www.ucl.ac.uk/disclaimer/"
      title="Disclaimer">Disclaimer</a> |
    <a href="http://www.ucl.ac.uk/accessibility/"
      title="Accessibility">Accessibility</a> |
    <a href="http://www.ucl.ac.uk/priv/" title="Privacy">Privacy</a>
  </div>
  {currentdate()}|datetime(custom,"%l %d %F %Y")}
</div>

<div id="logo">
  <form name="google_search" id="google-search" method="get"
    action="http://www.google.com/univ/ucl">
    
    <label for="q">Enter Search Text:</label>
    <input type="text" size="15" maxlength="255" name="q" id="q"
value=""
      tabindex="1" />
    <label for="sa"></label>
    <input type="submit" name="sa" id="sa" value="Go" tabindex="2"
/>
    <input type="hidden" name="sitesearch" id="sitesearch"
      value="ge.ucl.ac.uk" />
  </form>

  <a href="http://www.ucl.ac.uk" title="Main UCL Site">
    
  </a>
  <a href="/" title="GE Home Page">
    
  </a>
</div>

<div id="breadcrumbs">
  <ul>
    {section name=Path loop=$module_result.path}
    {section show=$Path:item.url}
    <li><a href={concat("/", $Path:item.url_alias)|ezroot}>
      {$Path:item.text|wash}</a>
      &#8250;&#8250;
    </li>
    {section-else}
    <li>{$Path:item.text|wash}</li>
    {/section}
  {/section}
</ul>
</div>
```

The reason why the breadcrumbs are placed in an unordered list () is because, effectively, they are a list of the elements in the path that defines the current location on the site. This is generally considered good practice and is more semantically correct than if each one were placed in a span or a div container. For the search box we use a label; this is considered a good practice when building accessible sites.

navigation.tpl

By using nested unordered lists, we can define a logical structure for the menu. This makes an enormous difference when viewed in a text browser, because the structure is much clearer than if we had used div or plain href tags.

It also allows us to use CSS styling to visually design the navigation. The li selector applies to all elements in the list, whether they are top level or in the sub-level. The selector li li will only apply to the sub-level, allowing us in this case to indent the links to demonstrate their structure in the navigation. This kind of navigation would have required some cunning using a table-based layout, probably using some invisible spacer GIFs or the like. Using CSS, we have a lean markup that can be transformed using some simple CSS to manipulate the layout as we please:

```
<div id="navigation">
  {let folder_list=fetch( content, list, hash(parent_node_id, 2,
sort_by,
    array( array( priority ) ) ) )}
  <ul>
    {section name=top_level loop=$folder_list}
    <li><a href={concat("/", $folder2:item.url_alias)|ezroot}>
      {$folder2:item.name|wash}</a>
      {section show=$top_level:item.node_id|
eq($module_result.path[1].node_id)}
      {let sub_folder_list=fetch( content, list, hash(parent_node_id,
$module_result.path[1].node_id,
sort_by, array( array( priority ) ) ) )}
      <ul>
        {section name=sub_level loop=$top_level:sub_folder_list}
        <li><a href={concat("/", $top_level:
sub_level:item.url_alias)|ezroot}>{$top_level:
sub_level:item.name|wash}</a></li>
        {/section}
      </ul>
    {/let}
    {/section}
  </li>
  {/section}
</ul>
{/let}
</div>
```

The following screenshot shows the breadcrumbs and navigation bar viewed in the **Lynx** text browser. You can see the clear navigational structure achieved as a result of using lists:

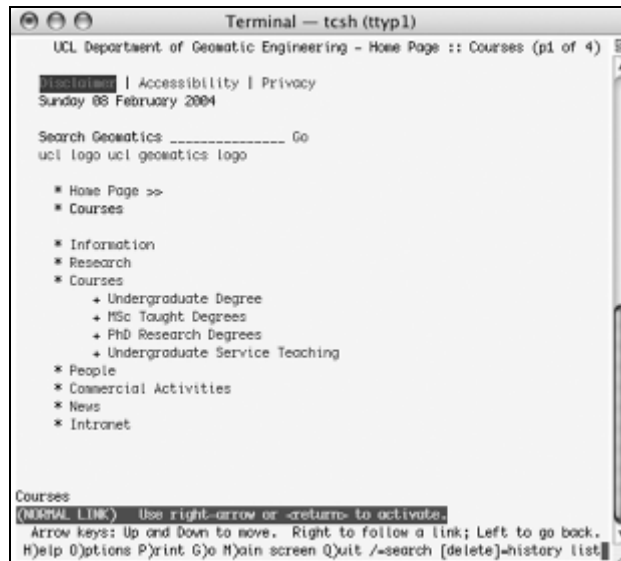


image.tpl

This template is slightly more complicated because of the variables that are available for use in the different templates. In templates like this one that are manually included (and not related to a particular node), the `$node` variable is not available. Instead, we have to use the `$module_result` variable if it is necessary to access data for the current node.

In our case it is necessary to use the image for the node, so we quickly fetch the object for that node and check if it has an image. If it does not have an image, the default image is used.

This website was created using eZ publish 3.1, but in version 3.3, the mechanism for showing images has changed quite dramatically.

```
<div id="image">
  {let mynode=fetch('content', 'node', hash(node_id,
$module_result.node_id))}
  {section show=$mynode.object.data_map.image.content}
  {attribute_view_gui
attribute=$mynode.object.data_map.image.content[side]}
  {section-else}
  {let default-side=fetch('content', 'node', hash(node_id, 605))}
  {attribute_view_gui attribute=$default-
side.data_map.image.content[side]}
  {/let}
  {/section}
{/let}
</div>
```

footer.tpl

Again in this file, because we do not have the `$node` variable, we must fetch the object for the node to reference the time it was last modified. The only other template code in this include file is the one that shows the current year, which passes the `currentdate()` function into the `datetime()` formatting function. Everything else in here is regular XHTML:

```
{let mynode=fetch('content', 'node', hash(node_id,
$module_result.node_id))}

<div id="footer">
  Department of Geomatic Engineering - University College London<br
/>
  Gower Street, London, WC1E 6BT | Tel: +44 (0)20 7679 2740 | Fax:
+44 (0)20
  7380 0453<br />

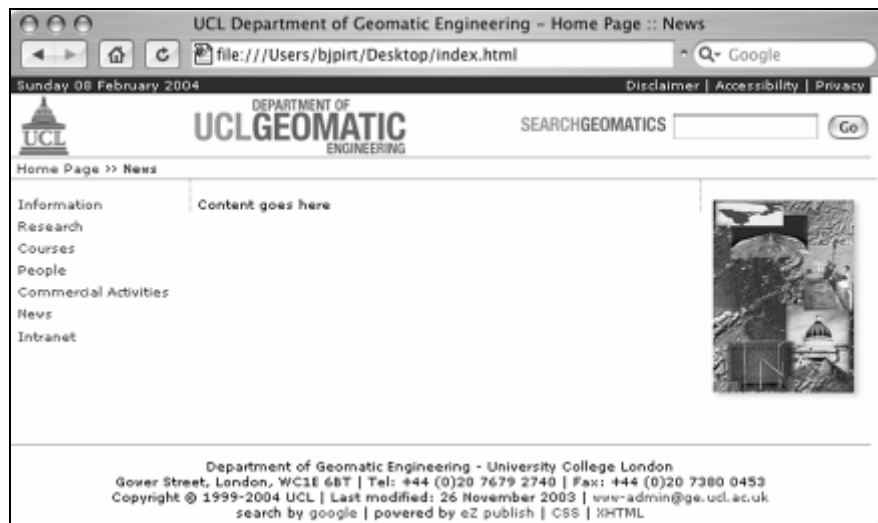
  Copyright &copy; 1999-{currentdate()}|datetime(custom,"%Y")} UCL |
  Last modified: {$mynode.object.modified|datetime(custom,"%d %F
%Y")} |
  <a href="mailto:www-admin@ge.ucl.ac.uk">www-
admin@ge.ucl.ac.uk</a><br />
  search by <a href="http://www.google.com/">google</a> |
  powered by <a href="http://ez.no">eZ publish</a> |
  <a href="http://jigsaw.w3.org/css-validator/validator">CSS</a> |
  <a href="http://validator.w3.org/check/referer">XHTML</a>
</div>

{/let}
```

Now that we have defined these basic template features, we can apply the CSS rules to the document to put it in shape. The following images show the difference between the site with and without the stylesheet.



This image shows the unstyled page—you can see that the information on the unstyled page still makes sense because a logical structure was used for the XHTML. The following image shows the document with the stylesheet applied:



CSS Rules

The rules that were used to perform this transformation are as follows:

```
body {
  margin:0px;
  padding: 0px;
  background-color: #FFF !important;
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-size:70%;
}
```

This rule sets the margin and padding for the document to zero and ensures that the background stays white with the `!important` declaration. This is a good place to specify the font used in the document as it will cascade down into all of the other elements and eliminate redundant style declarations. The font-size declaration is a little trick found at <http://www.thenoodleincident.com/tutorials/typography/> and is used to get around the problem associated with specifying font sizes and accessibility in Windows Internet Explorer.

The problem is this—accessibility rules suggest that the user should be able to resize the text on a page using the browser's built-in text-resizing feature. This presents a problem in IE when you specify the text size in pixels—the text will not resize. The other problem is that if you specify a font-size of say 0.7em, it will actually come out extremely small and not 70% of the user's default font size.

By specifying the font size as a percentage, we can leave the default text size alone, and scale down the font size of the whole document. Now we can resize the text in IE, while maintaining consistency across browsers.

The following rules for the date ID simply set background colors and float the relevant text to the right:

```
#date {
  padding-left:5px;
  background-color:#1C1C73;
  color: white;
}

#date .align-right {
  float:right;
  display:inline;
  margin-right:5px;
}

#date .align-right a {
  color: white;
  text-decoration: none;
}
```

The following rules lay out the logos and the Google search box in the correct places on the page:

```
#logo {
  background-color:#FFF;
  height:49px;
  border-bottom:1px solid #AAA;
  border-top:1px solid #1C1C73;
}
#logo a img.ucl-logo {
  margin-left:7px;
  display:inline;
  border:none;
}
#top-logo a img.ucl-geo-logo {
  margin-left:94px;
  display:inline;
  border:none;
}
#logo #google-search {
  position:absolute;
  right:0;
  display:inline;
  float:right;
  height:35px;
}
#logo #google-search input {
  margin-top:16px;
}
#logo #google-search img {
  padding-top:10px;
}
```

The following rules lay out breadcrumb elements and tell the browser to display the list elements as inline, which means that they will all flow on the same line as we want for our breadcrumbs:

```
#breadcrumbs {
  padding:2px 5px 2px 0;
  margin:0;
  width:100%;
  background-color:#FFF;
  border-bottom:1px solid #AAA;
}

#breadcrumbs ul {
  display:inline;
  padding:0;
  margin:0;
}

#breadcrumbs li {
  display:inline;
  padding:0;
  margin:0;
}
```

The set of rules outlined here positions the navigation on the page, and then, using multiple selectors, styles the menu items in a way that makes sense with the structure of the navigation:

```

#navigation {
  float:left;
  left:0;
  padding-top:13px;
  width:140px;
}

#navigation ul {
  list-style-type:none;
  margin:0;
  padding:0;
}

#navigation ul ul {
  margin-top:6px;
}

#navigation li {
  padding-left:6px;
  margin:0 0 5px 0;
  display:block;
}

#navigation li li {
  margin-left:13px;
  padding:0;
  display:block;
}

#navigation a {
  margin:0;
  padding:0;
  display:inline;
}

```

Because we have floated the navigation to the left-hand side, we set a left margin on the main content to cover this distance, effectively creating two columns. We do the same on the right-hand side to create the third column for the image. By adding borders we can delineate between these three columns.

```

#content {
  padding:13px 5px 0;
  margin-right:0 132px 0 140px;
  border-left:dotted 1px #AAA;
  border-right:dotted 1px #AAA;
}

#image {
  float:right;
  width:110px;
  padding: 0 7px 7px 0;
  margin-right:6px;
}

```

The image is floated right and overlaps the margin we set in the #content div tag.

Earlier, we mentioned bandwidth as a reason for using XHTML and CSS instead of table-based design. Because the template used by UCL utilizes tables, we are in a

position to make a direct comparison between the two techniques. The two bare templates were compared:

File	Size (in bytes)
UCL table-based template	9,842
XHTML template	4,769
CSS file	4,986
XHTML + CSS file	9,755

Well, a saving of 87 bytes, over a day that might even add up to 1KB of bandwidth savings. But that is not the whole story. First, the CSS file contains the styling for the entire site, and so only needs loading once. Secondly, it is also cached, which means that on subsequent page loads it will not need downloading again. This means that if a user looks at, say 20 pages, they will be using only 51% of what they would if they were downloading the table-based version. If a transitional approach were used, where tables were used for layout and stylesheets for the rest of the styling, we could expect to see bandwidth usage somewhere between these two scenarios.

This section has shown the planning that went into building the framework into which the content of the site is placed. The link between planning the parts of the layout, the structure of the templates, and the design of the CSS should be clear. By breaking the design of the site down into logical parts and using these to define our CSS, we can achieve a clear and consistent approach to the design of the site. We will take this one step further when we design the content that fits into the template we have just put together.

Designing the Content

The building block of our standards-compliant eZ publish site is the `div` tag. By encapsulating all of our objects into `div` tags with suitably named classes, we can gain a large degree of consistency between the internal structure of eZ publish and the external structure of the document. In the previous section of this case study, we saw how we can create a highly modular document structure through the use of `div` tags with IDs. In this section, we will continue this by aligning the classes of data types in eZ publish with the classes we create in the stylesheet.

We will look at one of the content classes to illustrate the way this process works and ensure consistency in all areas of the application.

The News Article Class

There are three parts to creating a new class:

- Defining the data class using eZ publish, where we decide upon the datatypes that the class should contain.
- Creating the templates for eZ publish to use when displaying the page. There may be multiple templates for each class, depending upon the different views of the data.
- Creating corresponding CSS rules to style the output.

The Data Class Definition

We saw the fields that were necessary for this class earlier, but here we will go into a little more detail:

Data Field	Datatype	Description
Title	Text Line (Required)	The title of the article
Intro	Text Line	The introduction to the article
Body	XML Text Field	The full story of the article
Valid	Checkbox	Whether the article is still valid or not

Once we have defined the data class using eZ publish, we can then look at creating the templates to output this information.

Class Templates

Overrides are one of the most powerful features of the eZ publish templating system. The main files that we will be overriding are `/node/view/full.tpl` and `/node/view/line.tpl`. These templates provide the two main views of our classes, and both are used when we use the templating function `node_view_gui()`. If you recall, the `full` template is normally used when we want to show a view of only one node, and the `line` template is used when we want to show a list of nodes on one page. Both templates show different views of the same information. In this case, the site only has the News Article class in one section so we can define a site-wide override for this class. If we wanted to show a News Article class with different formatting in different parts of the site, it would be necessary to define sections in eZ publish, and then to create overrides for the different sections.

For our site, we define the following overrides:

File: override.ini.append.php

```
[news_full]
Source=node/view/full.tpl
MatchFile=news_full.tpl
Subdir=templates
Match[class]=20
[news_line]
Source=node/view/line.tpl
MatchFile=news_line.tpl
Subdir=templates
Match[class]=20
```

These tell eZ publish to use the files `person_full.tpl` and `person_line.tpl` for the full and line views respectively.

news_full.tpl

This is a very simple view of the data that simply presents the title of the story as the header and emphasizes the introduction. The styles that will be applied to this are the same styles that are defined for the rest of the website. This illustrates one of the strong points of CSS; the ability to redefine existing HTML elements such as the `<em>` and `<h1>` tags. By defining overall styles for the site, a great degree of consistency can be achieved with very little effort. It also shows that not everything needs wrapping in a `<div>` tag.

```
<h1>{$node.name}</h1>
<em>{attribute_view_gui attribute=$node.object.data_map.intro}</em>
{attribute_view_gui attribute=$node.object.data_map.body}
{attribute_view_gui attribute=$node.object.data_map.valid}
```

news_line.tpl

This template just shows the title of the story, which acts as a link to the full story. It is all enclosed in one `<div>` so that we can style a surrounding box.

```
<div class="news-line">
  <a href="{concat("/", $node.url_alias)|ezroot}">
    {$node.data_map.title.data_text|wash}
  </a>
  <p>
    {attribute_view_gui attribute=$node.data_map.intro}
  </p>
</div>
```

CSS Rules

Now that we have the classes defined, we can create the CSS rules to style them.

```
.story
{
  border:solid 1px #CCC;
  background-color:#EFEFEF;
  margin-bottom:5px;
}
.story a
{
  display:block;
```

```

padding:1px 5px;
border-bottom:solid 1px #CCC;
}
.story p
{
display:block;
padding:1px 5px;
margin:0;
}

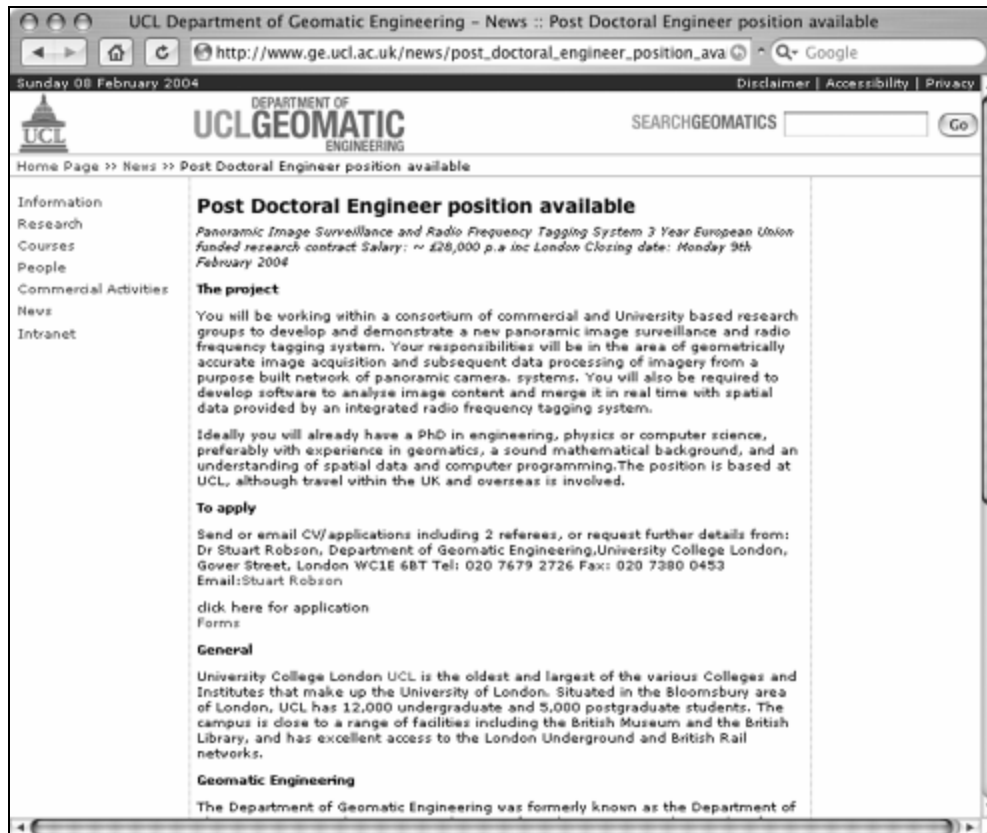
```

These will style the line view of the story so that there is a grey box around it as shown in the following image. The full view of the news story uses the global CSS rules for these elements.

The following image shows the line view of the News class:



And here we can see the full view of the News Class:



Performance

While the default settings for eZ publish are good for development, they are unsuitable for production use. To increase the performance of eZ publish, there are a number of things we can do:

- Enable view caching
- Enable template compiling
- Enable template caching
- Use cache blocks in the templates
- Install a PHP accelerator

View Caching

View caching takes the content of the override templates you have created and caches them so that all eZ publish needs to do is insert the correct data into the right places in the template. It is best to disable this during development because any changes you make to the templates will not be seen on the site until you flush the cache. View caching can be enabled with the following setting:

File: site.ini.append.php

```
[ContentSettings]
ViewCaching=enabled
```

Template Compiling

Template compiling involves compiling templates that use the templating language used by eZ publish into templates that use native PHP. This obviously speeds up the lengthy templating process quite considerably. Without this option enabled, the templating process takes up a large majority of the time it takes to output a page. To enable this setting, use the following configuration:

File: site.ini.append.php

```
[TemplateSettings]
TemplateCompile=enabled
```

Template Cache Blocks

View caching and template compiling will improve the performance of eZ publish with a relatively small amount of effort. Using cache blocks, however, takes a bit more effort but can also have the greatest rewards. Cache blocks are a way of caching the output of dynamic content in a template into a plain HTML file, which is then simply included in the output. In its simplest form, the cache block statement works as follows:

```
{cache-block}
  Lots of dynamic code here
{/cache-block}
```

However, if we want to apply this to something like our menu system or the breadcrumbs, this would be of no use because every page would be given the same menu or breadcrumbs. To get around this, we can use the keys parameter, which gives a unique identifier to each state of the cache block. For example, if we had some code that was different on every page, we could use a variable that is unique on each page, such as `$uri_string`, which would give us:

```
{cache-block keys=$uri_string}
  Lots of code here which is different on every page
{/cache-block}
```

Alternatively, if we have a site that changes very regularly, we can specify an expiry time in seconds like so:

```
{cache-block expiry=180}  
  Lots of code here which expires every 3 minutes  
{/cache-block}
```

So if we apply this knowledge to our main `pagelayout.tpl` template, we get the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
  
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">  
<head>  
  {cache-block}  
    {include uri="design:page_head.tpl"}  
  {/cache-block}  
</head>  
<body>  
  {include uri="design:header.tpl"}  
  {cache-block keys=$module_result.path[1].node_id}  
    {include uri="design:navigation.tpl"}  
  {/cache-block}  
  <div id="content">  
    {$module_result.content}  
  </div>  
  {cache-block keys=$module_result.node_id}  
    {include uri="design:image.tpl"}  
    {include uri="design:footer.tpl"}  
  {/cache-block}  
</body>  
</html>
```

This template caches in the following ways:

- `page_head.tpl` will be cached the same for every page.
- The navigation will be cached for two levels of navigation from the path variable.
- The image and footer will be different for every page.

Furthermore, within `header.tpl`, we would tell eZ publish to cache the date bar according to the current date, the logo and search box for every page, and the breadcrumbs according to the individual node.

PHP Accelerators

It is always a good idea to install a PHP accelerator if it is possible on your server. It often speeds the site up by a factor of 2. There are several accelerators that work well with eZ publish:

Cache	URL
Alternative PHP Cache	http://apc.communityconnect.com/
IonCube PHP Accelerator	http://www.phpaccelerator.co.uk/
Turck MMCache	http://turck-mmcache.sourceforge.net/

Of these, I have used both the IonCube PHP Accelerator (on the UCL site) and Turck MMCache with success. See individual sites for instructions on how to install these extensions to PHP.

Benchmarking

Benchmarking is a very good idea if you want to understand the effects your performance tweaks have had on the server. The simplest technique for measuring performance is a utility that comes with the Apache web server called **ab**.

ab

ab stands for the Apache Benchmarking tool and is used to test a server for the number of requests it can deal with in a period of time. It is used as follows:

```
ab -n 100 -c 4 www.ge.ucl.ac.uk/
```

This command means that it will make 100 requests (-n) with a concurrency of 4 (-c), and will report back with results that look like:

```
Server Software:      Apache/2.0.40
Server Hostname:      www.ge.ucl.ac.uk
Server Port:          80
Document Path:        /
Document Length:      6375 bytes
Concurrency Level:     4
Time taken for tests:  14.216923 seconds
Complete requests:    100
Failed requests:       0
Write errors:          0
Total transferred:    687000 bytes
HTML transferred:     637500 bytes
Requests per second:  7.03 [#/sec] (mean)
Time per request:     568.677 [ms] (mean)
Time per request:     142.169 [ms] (mean, across all concurrent
requests)
Transfer rate:        47.13 [Kbytes/sec] received
Connection Times (ms)
      min      mean[+/-sd] median    max
Connect:    0       0   0.0      0      0
Processing: 278     559 309.6    467    1955
Waiting:    223     535 311.0    462    1955
Total:      278     559 309.6    467    1955
Percentage of the requests served within a certain time (ms)
 50%      467
```

```

66%    578
75%    678
80%    709
90%    953
95%    1226
98%    1629
99%    1955
100%   1955 (longest request)

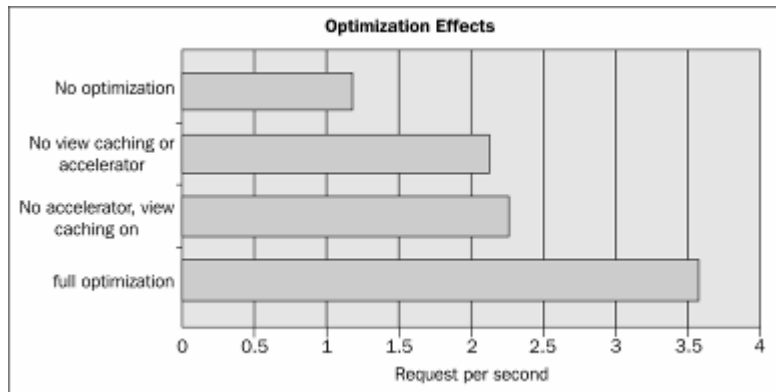
```

From this output we can see that this setup can serve 7 pages per second with this level of concurrency. This is suitable for a site that is not extremely busy and the loading time of approximately half a second is small enough to make the site feel quite responsive. With less concurrency, the response time becomes faster as shown here:

Concurrency	1	2	3	4
Requests per Second	3.58	6.88	6.98	7.04
Time per Request (mean)	279.547ms	290.838ms	429.864ms	567.885ms
Time per Request (mean, all requests)	279.547ms	145.419ms	143.288ms	141.971ms

Effects of Optimization

Having covered various ways of improving the performance of your eZ publish installation and then looking at how to measure them, you can now see their effects, as measured by ab, in the following graph:



Resources

Resource	Where to find it
Designing with Web Standards – Jeffrey Zeldman	ISBN: 0-7357-1201-8
CSS: The Definitive Guide – Eric Meyer	ISBN: 1-56592-622-6
Eric Meyer on CSS	ISBN: 0-7357-1245-X
CSS Discussion list on Incutio	http://css-discuss.incutio.com
Glish resources on CSS	http://www.glish.com/css/
Maxdesign CSS tutorials	http://css.maxdesign.com.au/index.htm
A List Apart	http://www.alistapart.com/
The W3C	http://www.w3.org

Summary

In this chapter, we began with a brief introduction to web standards and their real-world benefits. We then looked at the design of a site; beginning with the data classes that are needed and their CSS class counterparts.

We saw the design decisions behind the layout of the overall site template and the eZ publish components that are behind it. We then looked at maintaining a consistency between standards-based design and the underlying structure of eZ publish. Using one of the data classes as an example, we followed the design process from defining the data class to writing the CSS classes for its display.

After building the site, we went on to examine ways of speeding it up using various optimizations, such as PHP accelerators and caching, and their settings in eZ publish. Through the use of benchmarking, we were able to test the performance of our site and determine the effect these optimizations had. You should now understand the thought process behind the design of this site and hopefully be able to apply this to future sites of your own.



Template Operators and Functions

Operators

Operators are used to modify the content for presentation or to perform other useful manipulations. The coming sections discuss the available functions and operators in detail, per category. The categories are:

- String operators
- Array operators
- Mathematical operators
- Localization operators
- Miscellaneous

Operators modify content by using the '|', much like UNIX pipes. For example:

```
{"Hello"|count} {*results in 5 *}
```

Operators can take one or more optional parameters, which are put in parentheses and separated with a comma.

String Operators

A general note for Unicode-enabled sites: consult the documentation on <http://www.php.net> regarding `mb_string` and the configuration options in `php.ini`. This allows for Unicode-safe operations for most (but not all) of the string functions mentioned here. Note that the operators `upcase`, `downcase`, `upfirst`, and `upword` require the server or PHP set to the correct locale to work properly. eZ publish doesn't support proper Unicode case mapping yet, so you may get strange results with a wrong locale.

The following table gives an alphabetical overview of string operators and their functions:

Operator	Purpose
append	Appends element(s) to a string.
begins_with	Checks if a string starts with a specific element/sequence.
break	Replaces newlines with XHTML line breaks (same as <code>n12br</code>).
chr	Creates a string based on an array of ASCII/Unicode values.
compare	Compares the contents of two strings.
concat	Concatenates values to one string.
contains	Checks if a string contains a specific element.
count_chars	Counts and returns the number of characters (string length).
count_words	Counts and returns the number of words within a string.
crc32	Calculates the CRC32 polynomial of a string.
downcase	Converts all alphabetical characters to lowercase.
ends_with	Checks if a string ends with a specific element/sequence.
explode	Splits a string into an array of substrings.
extract	Extracts a portion from a string.
extract_left	Extracts a portion from the start of a string.
extract_right	Extracts a portion from the end of a string.
indent	Indents a string by inserting characters at its start.
insert	Inserts stuff at specified position in a string.
md5	Calculate the md5 hash of a string.
mimetype_icon	Converts a mimetype string into an icon. (from version 3.4)
n12br	Converts newlines to HTML <code>
</code> (see also the <code>break</code> operator)
ord	Returns characters in a desired sequence.
pad	Makes sure a string is at least n characters long.
prepend	Prepends a custom sequence to a string.
remove	Removes element(s) from a string.
repeat	Repeats the contents of a string.
reverse	Reverses the contents of a string.
rot13	Performs a rot13 transform on a string.

Operator	Purpose
<code>shorten</code>	Shortens a string to a few characters and adds trailing sequence.
<code>simplify</code>	Transforms multiple consecutive characters into one.
<code>trim</code>	Strips whitespace from the beginning and/or end of a string.
<code>upcase</code>	Converts all alphabetical characters to uppercase.
<code>upfirst</code>	Converts the first character to uppercase.
<code>upword</code>	Converts the first character of all words to uppercase.
<code>wash</code>	General text wash. Translates potentially bogus strings to friendly ones.
<code>wordtoimage</code>	Replaces special sequences of text with images.
<code>wrap</code>	Wraps text to lines at a specified length.

String Transformations

The majority of these operators actually transform strings in one way or another.

Changing the Case of Characters

The following examples illustrate the changing of case:

```
{"Making me shout!"|upcase}
```

This returns the string "MAKING ME SHOUT!"

```
{"STOP MAKING ME SHOUT!"|downcase}
```

This returns the string "stop making me shout!"

```
{'you are invited to my party!'|upfirst}
```

This returns the string "You are invited to my party!"

```
{'commodore amiga user'|upword}
```

This returns the string "Commodore Amiga User"

Inserting Breaks, Shortening, Padding, or Trimming Text

For display purposes, text can be transformed using the `break` (or `nl2br`), `trim`, `wash`, `simplify`, `wrap`, and `shorten` operators. For example:

```
{"The next summer  
conference will take  
place in June  
"|break}
```

This returns "The next summer
conference will take
place in June
"

Stripping whitespace from the start and end of a string is done using the `trim` operator.

In spite of what the online docs say, you have to provide an argument like whitespace (tested eZ publish 3.3-3), when using this operator.

For example:

```
{ "    I want you to want me  "|trim(' ' ) }
```

This returns the string "I want you to want me".

For reducing spaces *inside* strings, use the `simplify` operator. You can also reduce duplication of characters within strings. For example:

```
{ "aabcaabcaaaabc"|simplify('a') }
```

This returns the string "abcabcabc". Duplicate instances of the letter 'a' have been trimmed to one.

To wrap text to limit the width of text blocks, you need to use the `wrap` operator. The `wrap` operator needs three parameters, as in `wrap(width, break_sequence, cut)`. By inserting the `break_sequence` characters into `$string`, the `wrap` operator wraps given text at a specified width. All parameters are required (version 3.3-2). The `break_sequence` parameter specifies a string containing the desired break sequence to use; the default is `\n` (newline). The `cut` parameter specifies whether the string should be wrapped at the specified width. In other words, if you have a word that is larger than the given width, it is broken apart. For example:

```
{ "Good morning fellow eZ publish user withalongname"|wrap(8, '-', true()) }
```

This returns the string "Good-morning-fellow-eZ-publish-user-withalon-gname".

Now consider:

```
{ "Good morning fellow eZ publish user withalongname"|wrap(8, '-', false()) }
```

This returns "Good-morning-fellow-eZ-publish-user-withalongname".

`wash` is a general character/string washing operator. The `washtype` parameter specifies the washing type: e-mail, PDF, or XHTML (default). If you have a string that contains bogus characters (or sequences) that could alter the intended result, you should 'wash' it before outputting it. For example:

```
{ "A nasty string with a <br>"|wash }
```

This returns "bogus string
".

```
{ "bf@ez.no"|wash("email") }
```

This returns "bf[at]ez[dot]no".

The conversion of e-mail characters is controlled by the ini file settings [WashSettings] in `template.ini(.append)`

The shorten operator shortens string to `maxLength` characters and adds a trailing ellipsis sequence. Note that `maxLength` also includes the length of the trailing sequence. If the input string is shorter than `maxLength`, it will not be shortened. For example:

```
{"This string is too long and needs to be shortened!"|shorten(17)}
```

This returns the string "This string is...".

The pad operator makes sure that the input string is at least `length` characters long by inserting extra characters at the end (padding). It is possible to specify what character to use with the `pad_char` parameter. The default pad character is a space. For example:

```
{"Too short!"|pad(16)}
```

This returns the string "Too short! "

```
{"Too short!"|pad(16, "-")}
```

This returns the string "Too short!-----"

Joining, Extracting, and Further Manipulation of Text Portions

The concat operator can be used to join an arbitrary number of strings (or any other values that are implicitly converted to strings). For example:

```
{concat("Good ", " morning" , " Norway")}
```

This returns the string "Good morning Norway".

You can also use the `implode` operator (actually an array operator) to join words, as explained further in Chapter 3. This also concatenates strings with an optional separator.

The append and prepend operators are useful alternatives when you need to add text to the beginning or end of a string. The repeat operator is used to repeat text patterns, for example:

```
{"My way"|append ( "... " )| repeat (2)}
```

This returns "My way... My way..."

The `extract`, `extract_left`, and `extract_right` operators extract text portions from a given string. The returned portion is defined by the `offset` and `length` parameters. If `length` is omitted, the rest of the string from the offset will be returned. For example:

```
{"A new car"|extract(2,7)}
```

This returns the string "new car".

```
{"Hello sun!"|extract_left(5)}
```

This returns the string "Hello".

Counting and Comparing Strings

The `count_words` operator simply counts and returns the number of words that are found within the inputted string. For example:

```
{ "How many times did you try to install eZ publish" | count_words }
```

This returns an integer with its value set to 10.

The `count_chars` operator counts and returns the number of characters (string length). For example:

```
{ "General protection fault!" | count_chars }
```

This returns an integer with its value set to 25.

You can also use the general count operator, which behaves exactly as the `count_chars` operator for strings.

In general, strings can be compared using the standard boolean operators, but the eZ publish template engine provides a few additional ones. The `begins_with` and `ends_with` operators check for the existence of a substring at the beginning and end of strings. For example:

```
{ 'Linux is great!' | ends_with('great!') }
```

This returns `true()`.

```
{ 'Linux is great!' | begins_with('great') }
```

This returns `false()`.

The `contains` operator can be used as a substring match operator, for example:

```
{ 'A nice phrase' | contains('nice') }
```

This returns `true()`.

```
{ 'A long phrase is to avoid' | contains('short') }
```

This returns `false()`.

Array Operators

The array constructors `array` and `hash` are not repeated here. The following table lists the available array functions:

Operator	Purpose
<code>append</code>	Appends element(s) to an array.
<code>array</code>	Builds an array using specified elements.
<code>begins_with</code>	Checks if an array starts with a specific element/sequence.

Operator	Purpose
compare	Compares the contents of two arrays.
contains	Checks if an array contains a specific element.
counts	Counts the number of array elements
ends_with	Checks if an array ends with a specific element/sequence.
explode	Splits an array into an array of sub-arrays.
extract	Extracts a portion from an array.
extract_left	Extracts a portion from the start of an array.
extract_right	Extracts a portion from the end of an array.
hash	Creates a hash (associative array).
implode	Joins array elements with strings (see previous section).
insert	Inserts an element/sequence at specified position in array.
merge	Merges input and passed arrays into one array.
prepend	Adds element(s) to start of an array.
remove	Removes element(s) from an array.
repeat	Repeats the contents of an array.
reverse	Reverses the contents of an array.
array_sum	Calculates the sum of values in an array.
unique	Removes duplicate values from an array.

The merge operator will merge the `$input_array` variable with two or more arrays passed as parameters into a single array (the values of an array are appended to the end of the previous array). It returns the resulting array. For example:

```
{array(7,2)|array_merge(array(6,3),array(5,4))}
```

This returns an array with content: (7,2,6,3,5,4).

The unique operator removes duplicate elements from an array, for example:

```
{array(1,1,1,3,4,4,5)|unique}
```

This returns the following array: (1,3,4,5).

The `array_sum` operator calculates the sum of values in an array and returns the value (integer or float), for example:

```
{array(13,4,8)|sum}
```

This returns an integer with its value set to 25.

The `implode` operator returns a string containing a string representation of all the array elements (from the inputted array) in the same order, with the *separator* string between each element, for example

```
{array('tree','flower')|implode(' | ')}
```

This returns the string "tree | flower".

With arrays, the `explode` operator expects the parameter to be an integer offset, specifying where to split the array in two. It will return an array containing the two resulting arrays, for example:

```
{array('f','g','g','th','e')|explode(3)}
```

This returns the following array: (`array('f','g','g')`, `array('th','e')`).

The `extract` operator will return a part of an input array. The returned portion is defined by the offset and length parameters. If length is omitted, the rest of the array from the offset (remember array indices start at 0) will be returned. For example:

```
{array(a,b,c,e)|extract(2)}
```

This returns the array (e).

```
{array(1,2,3,4,5,6,7)|extract(3,3)}
```

This returns the array (4,5,6).

The `extract_left` operator extracts a portion from the start of an array or a string. The length parameter defines the desired portion:

```
{array(1,2,3,4,5,6,7)|extract_left(3)}
```

This returns the array (1,2,3).

The `extract_right` operator extracts a portion from the end of an array; the length parameter defines the desired portion.

For example:

```
{array(a,b,c,d,e)|extract_right(3)}
```

This returns the array (c,d,e).

The `begins_with` operator checks if the array begins with a specified element/sub-array. It returns `true()` or `false()`.

For example:

```
{array(1,2,3,4,5,6,7)|begins_with(1,2,3)}
```

This returns `true()`.

```
{array(1,2,3,4,5,6,7)|begins_with(2,3,4)}
```

This returns `false()`.

The `ends_with` operator checks if the input array ends with a specified element/. It returns `true()` or `false()`.

For example:

```
{array(1,2,3,4,5,6,7)|ends_with(5,6,7)}
```

This returns `true()`.

```
{array(1,2,3,4,5,6,7)|ends_with(4,5,6)}
```

This returns `false()`.

The `repeat` operator returns a repeated version of an array. The `times` parameter defines the number of times the array should be repeated. The result returned is the array. For example:

```
{array(a,b)|repeat(2)}
```

This returns the array `(a,b,a,b)`.

The `reverse` operator simply returns a reversed version of an inputted array. For example:

```
{array(1,2,3)|reverse}
```

This returns the array `(3,2,1)`.

This operator `insert` inserts a sequence of elements at a specified position in an array. The position is counted from 0 for arrays and 1 for strings. For example:

```
{array(1,2,3,6,7)|insert(3,4,5)}
```

This returns the array `(1,2,3,4,5,6,7)`.

```
{array(a,b,c,d)|insert(2,array(g,h))}
```

This returns the array `(a,b,array(g,h),c,d)`.

The `remove` operator simply removes element(s) from an array or string and returns the chopped version of the array. The `offset` parameter defines the start of the portion to be removed while the `length` parameter defines the length of the portion. Note that position is counted from 0 for arrays and 1 for strings. For example:

```
{array(a,b,c,d,e,f)|remove(3,3)}
```

This returns the array `(a,b,c)`.

The `append` and `prepend` operators add element(s) to the end or beginning of arrays, as shown in the following examples:

```
{array(a,b)|append(c,d)}
```

This returns the array `(a,b,c,d)`.

```
{array(c,d)|prepend(a,b)}
```

This returns the array (a,b,c,d).

The compare operator simply compares the contents of two arrays and returns `true()` if they are the same or `false()` if they are not the same.

For example:

```
{array(a,b)|compare(array(c,d))}
```

This returns `false()`.

```
{array(a,b,c)|compare(array(a,b,c))}
```

This returns `true()`.

Mathematical Functions

The available mathematical functions are briefly described in this section.

sum and sub

`sum` returns the sum of all parameters, including the input parameter (if used), while `sub` subtracts all extra parameters from the first parameter. For example:

`sum( $a, $b, $c, $d )` results in `$a + $b + $c + $d`;

`$a|sum( $b, $c, $d )` results in `$a + $b + $c + $d`;

`sub( $a, $b, $c )` results in `$a - $b - $c`.

If an input parameter is supplied, it will be considered the first element and the parameters the subtractors. For example,

`$a|sub( $b, $c, $d )` results in `$a - $b - $c - $d`.

inc and dec

Increases or decreases either the input value or the first parameter by one. For example:

`{set i=0|inc}` will set `$i` to 1

and

`{set i=dec( $i )}` if used after the previous `{set}` will decrement `$i` to 0 again.

div

Divides the first parameter by all extra parameters. For example:

`div( $a, $b, $c )` results in `$a / $b / $c`.

If an input parameter is supplied, it will be considered the dividend, and the parameters the divisors. For example:

`$a|div($b, $c, $d )` equals `$a / $b / $c / $d`.

mod

Returns the modulus of the first input parameter divided by the second, for example:

`mod( 5, 3 )` returns 2.

mul

Multiplies all parameters and returns the result. If an input parameter is supplied, it will be included in the multiplication, for example `mul( $a, $b, $c )` results in `$a * $b * $c` and `$a|mul( $b, $c )` results in `$a * $b * $c`.

Max and min

Returns the largest or smallest value of all numeric parameters. Strings are evaluated as 0. For example:

`{max(1, 'tree', 'four', 5, ' a long string')}` returns 5;

`{min(1, 'tree', 'four', 5, ' a long string')}` returns 0;

`{min(-3,3,5)}` returns -3.

abs

Returns a positive value of either the input value or the first parameter. Strings are evaluated as 0 but quotes may be used around a legitimate number. For example:

`{abs(-3)}` and `{abs(' -3')}` and `{abs(' -three')}` results in "3 and 3 and 0".

ceil and floor

`ceil` returns the next highest or lowest integer value by rounding up the input value, if necessary. For example, `{ceil(3.14)}` results in 4.

`floor` returns the next lowest integer value by rounding down the input value, if necessary. For example, `{floor(3.14)}` results in 3.

round

Returns the rounded value of the input value. For example, `{round(3.14)}` results in 3 and `{round(8.8)}` results in 9.

Localization and Translation Operators

Localization operators and translation operators are useful for multilingual sites as well as for transforming certain values into the appropriate output format.

`datetime` is used for converting date or datetime values to an output format that you desire. You can define and use presets from `datetime.ini` or specify a custom format inline by using the special `custom` keyword.

The `l10n` operator can also be used for dates and times, and can also format numbers and currencies. The currently supported parameters are:

- `time`
- `shorttime`
- `date`
- `shortdate`
- `datetime`
- `shortdatetime`
- `currency`
- `clean_currency`
- `number`

The `l18n` operator is used to mark strings for translation. The translation can be in different languages and character sets. Optional arguments are the context in which the translation occurs, a more descriptive comment of what the string actually means (this helps translators) and a hash, which contains entries for substitutions that are needed in the translation of a string.

For example:

```
{let email='myname@mycompany.com'}
  {"Please contact %email about this"|l18n(
    'design/standard/footers', '', hash( '%nemail', $email ) )}
{/let}
```

The strings marked for translation can be translated by a tool called **Qt Linguist**. See the eZ systems website for more.

The `si` operator handles unit display of certain values, mostly technical and scientific units. It expects a unit parameter and an optional prefix. The available unit types are meter, gram, second, ampere, kelvin, mole, candela, byte, and bit. The available prefixes are binary, decimal, none, and auto. Other units and prefixes can be specified in `units.ini(.append.php)`.

For example,

```
{12|si(meter, milli)}
```

This returns "12,000.00 mm".

In previous versions of eZ publish, there used to be a `x18n` operator. Don't use it anymore: it is deprecated.

Logical Operators

ne

Returns `true` if one or more of the input parameters does not match. Matching is casual, meaning that an integer of value 0 will match a boolean of type `false`.

lt

Returns `true` if the input value is less than the first parameter. For example:

```
{1|lt(2)}
```

This returns `true`.

gt

Returns `true` if the input value is greater than the first parameter. For example:

```
{2|gt(1)}
```

This returns `true`.

le

Returns `true` if the input value is less than or equal to the first parameter. For example:

```
{1|le(1)}
```

and

```
{1|le(2)}
```

Both return `true`.

ge

Returns `true` if the input value is greater than or equal to the first parameter. For example:

```
{1|ge(1)}
```

and

```
{2|ge(1)}
```

Both return `true`.

eq

Returns `true` if the input value is equal to the first parameter. If no input value is available, it returns `true` if all parameters are equal. For example:

```
{1|eq(1)}
```

This returns true.

```
{eq(1,true(),false())|not,0|inc}}
```

This returns true.

null

Returns true if the input value is null, which is not the same as 0. For example:

```
{0|null()}}
```

This returns false.

not

Returns true if the input value is false. For example:

```
{false()|not()}}
```

This returns true.

true

Creates a true boolean. Remember to use brackets, for example {true()}.

false

Creates a false boolean. Remember to use brackets, for example {false()}.

or

Evaluates all parameter values until one is found to be true, and then returns that value. The remaining parameters are not evaluated. If there are no parameters or all elements are false, it returns false. For example:

```
{or(false(),false(),true(),false())}
```

This returns true.

You can also use other values than just true() or false() in the parameter list. For example:

```
{let count1=false() count2=5 count3=0}  
{or( $count1, $count2, $count3 )}  
{/let}
```

This will return 5 (\$count2).

and

Evaluates all parameter values until one is found to be false, and then returns that value. The remaining parameters are not evaluated at all. If there are no parameters, it returns false, and if no elements were false, it returns the last parameter value. For example:

```
{and(false(), false(), true(), false())}
```

This returns `false`.

choose

Uses the input count to pick one of the parameter elements. The input count equals the parameter index. For example:

```
{0|choose("a", "b", "c")}
```

This returns `"a"`.

contains

Returns `true` if the first parameter value is found in the input value, which must be an array.

Type Checking

These operators generally correspond to the PHP functions of the same name, where they exist.

Operator	Description
<code>is_array</code>	Returns <code>true</code> if the input or the first parameter is an array. If both input and parameter are supplied, the parameter will be used.
<code>is_boolean</code>	Returns <code>true</code> if the input or the first parameter is a boolean (<code>true</code> or <code>false</code>). If both input and parameter are supplied, the parameter will be used.
<code>is_integer</code>	Returns <code>true</code> if the input or the first parameter is an integer. If both input and parameter are supplied, the parameter will be used.
<code>is_float</code>	Returns <code>true</code> if the input or the first parameter is a floating-point number. If both input and parameter are supplied, the parameter will be used.
<code>is_numeric</code>	Returns <code>true</code> if the input or the first parameter is a number or a numeric string (a string consisting of numbers). If both input and parameter are supplied, the parameter will be used.
<code>is_string</code>	Returns <code>true</code> if the input or the first parameter is a string. If both input and parameter are supplied, the parameter will be used.

Operator	Description
<code>is_object</code>	Returns <code>true</code> if the input or the first parameter is an object (as opposed to a simple type like integer or float). If both input and parameter are supplied, the parameter will be used.
<code>is_class</code>	Returns <code>true</code> if the input or the first parameter is a class. If both input and parameter are supplied, the parameter will be used.
<code>is_null</code>	Returns <code>true</code> if the input or the first parameter is <code>null</code> . Warning: <code>null</code> is not the same as the integer <code>0</code> .
<code>is_set</code>	Returns <code>true</code> if the first parameter is <code>true</code> . <code>is_set</code> does not take an input.
<code>is_unset</code>	Returns <code>true</code> if the first parameter is <code>false</code> . <code>is_unset</code> does not take an input.
<code>get_type</code>	Returns the type of the input or the first parameter as a string. If both input and parameter are supplied, the parameter will be used. If the data is an object, then the string "object" and the class name will be returned. If the data is an array, then the string "array" and the array count will be returned. If the data is a string, then the string "string" and the string length will be returned. This function is mainly used for debugging and should not be used for comparisons.
<code>get_class</code>	Returns the class of the input or the first parameter as a string. If both input and parameter are supplied, the parameter will be used. If the data is not an object, <code>false</code> will be returned.

Image Handling

The template engine of eZ publish includes a powerful image sub-system. You can manipulate existing images for special effects or even create new images based on sets of existing images and images generated from text strings.

The image operators depend on an external image system, either ImageMagick or GD. This image system has to be installed and made available for eZ publish. Currently, three template operators are available for generating, manipulating, and displaying images. The `imagefile` and the `texttoimage` operators are used to generate images. The `imagefile` operator simply loads an image from the file system. The `texttoimage` operator creates an image of some text using a TrueType font. Both these operators return an image layer.

The layers generated by these two operators can be merged in various ways using the `image` operator, which returns the actual image object.

image

This operator creates and returns an image object. The operator basically flattens (merges) the image layers that were specified as parameters of the `image` operator. However, a parameter can be something else than an image layer. A parameter to this operator can be one of the following:

- A string
- An image layer
- An array

If a parameter is a string, the contents of the string will be used as the alternative image text for the image object that is returned by the operator (the `alt` XHTML tag).

If a parameter is an image layer, the layer will simply be merged into the image object that is to be returned by the operator.

If a parameter is an array, the operator will assume that the first array element (element zero) is the image itself (an image layer created by either the `imagefile` or the `texttoimage` operator), and that the second element is a hash (associative array) containing parameters for that layer. The following parameters can be used:

Parameter	Description
<code>transparency</code>	Float value ranging from 0 to 1.0 (0 = 0% and 1 = 100%)
<code>halign</code>	Horizontal alignment; use left, right, or center
<code>valign</code>	Vertical alignment; use top, bottom, or center
<code>x</code>	Absolute placement (works with left and right align)
<code>y</code>	Absolute placement (works with top and bottom align)
<code>xrel</code>	Relative placement; float value ranging from 0 to 1.0 (works with left and right align)
<code>yrel</code>	Relative placement; float value ranging from 0 to 1.0 (works with top and bottom align)

The `x` and `xrel` parameters cannot be used at the same time. The same goes for the `y` and the `yrel` parameters.

When right or bottom alignment is used, the coordinate system will shift to accommodate the alignment. This is useful for alignment and placement, since the placement is relative to the current coordinate system. Right alignment will start the axis at the right (0) and go on to the left (width).

Bottom alignment will start the axis at the bottom (0) and go on to the top (height).

If the operator is called directly, the `/design/standard/templates/image/imageobject.tpl` template will be used to display the image object that is returned by the operator. It is possible to override this template using the template override system.

The following code merges two images into one:

```
{image( imagefile( 'image1.png'|ezimage ),
        imagefile( 'image2.png'|ezimage ) )}
```

What happens here is that `image1.png` and `image2.png` are acted upon by the `ezimage` operator. This is simply done in order to prepend a valid path to the filenames. The `imagefile` operator is used to load the images (now with the correct path) from the file system. This operator returns an image layer. Finally, the `image` operator is used to merge these images (layers) together into one single image. The `image` operator will return an image object. In this case, the template system will insert the `/design/standard/templates/image/imageobject.tpl` template in order to display the image (you can use override templates for the default template as explained further in Chapter 3).

The following template code will create an image object with 70% transparent text, aligned in the bottom right corner of the church image fetched from the `var` directory. The alternative image text is set to `airplane`.

```
{image( "airplane",
        imagefile( 'var/cache/texttoimage/airplane.jpg' ),
        array( 'Let's fly'|texttoimage,
              hash( transparency, 0.7,
                  halign,      right,
                  valign,     bottom) ) )}
```

imagefile

This operator simply loads an image from the file system and returns it as an image layer. The layer can be used as a parameter to the `image` operator, which basically takes care of merging images.

For each image layer, the following information can be retrieved:

Attribute	Type
filepath	string
filename	string
width	integer
height	integer
alternative_text	string
imagepath	string
has_size	boolean

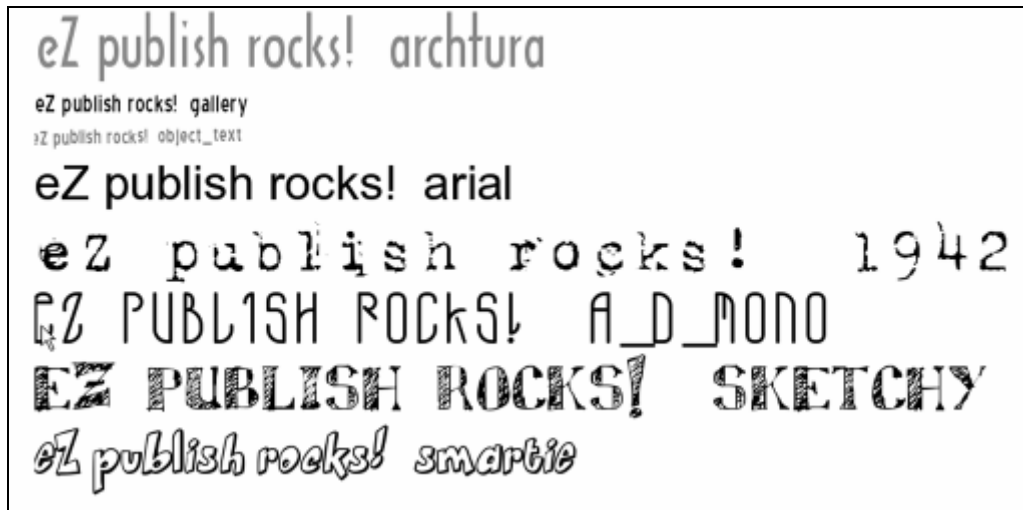
If the `has_size` parameter is set to `false`, the width and height parameters will contain the value `false` instead of zero-valued integers.

If the operator is called directly, the specified image will be displayed using a template associated with the image layer. This is the `/design/standard/image/layer.tpl` template. It is possible to override this template using the template override system.

Finally, an example of the default (eZ publish 3.3) definitions for image variations:

```
{'eZ publish rocks!  archtura'|texttoimage('archtura')}} <br />
{'eZ publish rocks!  gallery'|texttoimage('gallery')}} <br />
{'eZ publish rocks!  object_text'|texttoimage('object_text')}} <br />
{'eZ publish rocks!  arial'|texttoimage('arial')}} <br />
{'eZ publish rocks!  1942'|texttoimage('1942')}} <br />
{'eZ publish rocks!  a_d_mono'|texttoimage('a_d_mono')}} <br />
{'eZ publish rocks!  sketchy'|texttoimage('sketchy')}} <br />
{'eZ publish rocks!  smartie'|texttoimage('smartie')}} <br />
```

This produces the following result:



Other Template Operators

Let's now move on to other operators for the template.

count

Returns the "count" of the input value. The result depends on the type of the input value. The following table summarizes its effect.

Input type	Result of count
array	the number of elements in the array
object	the number of object attributes
string	the string length
numeric	the value of the numerical
boolean	0 for false(), 1 for true()
other	0

Examples:

```
{array( 1, 3, 5, 7 )|count} returns 4;
```

```
{'The quick brown fox'|count} returns 19.
```

Accessing Variables in the ini Files

ini variables can be read with the ezini operator. It takes three arguments:

- A section in the ini file (as put between [] brackets)
 - The variable itself
 - Optional: the ini file, if other than site.ini
- ```
{ezini('[section]','[variable]','[optional ini file]')}
```

For example:

```
{ezini('SiteAccessSettings','AnonymousAccessList')}
```

This returns an array of the modules and functions that do not require a real user logged in.

You are not limited to the standard eZ publish ini files. You could also create your own site specific ini files and sections to store design and layout parameters. This is handy if you need such parameters in different templates.

## cond

The cond operator expects an array with pairs of a condition and a value that will be returned upon a matching value of true.

For example:

```
{cond(false(),'red', true(),'blue')}
```

This returns the string "blue"

## **first\_set**

The `first_set` operator expects an array of values and/or variables and will return the first that is set.

For example:

```
{let $rootnode=2}
{first_set($myvar, $rootnode)}
{/set}
```

This returns `$rootnode` with a value of 2 unless `$myvar` is an existing variable and has a value assigned.

## **ez publish Kernel Operators**

### **ezurl**

Makes sure that the URL works for both virtual hosts and non-virtual host setups. For the latter, this means the inclusion of `index.php` in the URL, together with `hostname` and `siteaccess`.

### **ezroot**

This is the same as `ezurl`, but will omit the `index.php` from the url, regardless of your server configuration mode.

### **choose**

Uses the input as an index to pick an element from the array parameter. For example:

```
{1|choose('america', 'russia', 'china')} returns "russia".
```

### **contains**

Returns true if the first parameter value is found in the input value, which must be an array. Currently it works the same way as the PHP function `in_array()`. For example:

```
{array(1,3,5,7)|contains(4)} returns false().
```

### **currentdate**

The `currentdate` operator returns the current timestamp. For display purposes, you can feed this value through the `datetime` operator:

```
{currentdate()| datetime('custom','%Y-%m-%d %H:%m')}
```

This returns the current date and time.

For custom displays, you can use the following format characters:

| Format Character | Description                                                        |
|------------------|--------------------------------------------------------------------|
| %a               | Lowercase Ante meridiem and Post meridiem                          |
| %A               | Uppercase Ante meridiem and Post meridiem                          |
| %B               | Swatch Internet time                                               |
| %d               | Day of the month, 2 digits with leading zeros                      |
| %D               | A textual representation of a day, three letters                   |
| %F               | A full textual representation of a month, such as January or March |
| %g               | 12-hour format of an hour without leading zeros                    |
| %G               | 24-hour format of an hour without leading zeros                    |
| %h               | 12-hour format of an hour with leading zeros                       |
| %H               | 24-hour format of an hour with leading zeros                       |
| %i               | Minutes with leading zeros                                         |
| %I               | Whether or not the date is in daylight savings time                |
| %j               | Day of the month without leading zeros                             |
| %L               | A full textual representation of the day of the week               |
| %L               | Whether it's a leap year                                           |
| %m               | Numeric representation of a month, with leading zeros              |
| %M               | A short textual representation of a month; three letters           |
| %n               | Numeric representation of a month, without leading zeros           |
| %O               | Difference to Greenwich time (GMT) in hours                        |
| %r               | RFC 2822 formatted date                                            |
| %s               | Seconds, with leading zeros                                        |
| %S               | English ordinal suffix for the day of the month; 2 characters      |

---

| Format Character | Description                                                                                                                        |
|------------------|------------------------------------------------------------------------------------------------------------------------------------|
| %t               | Number of days in the given month                                                                                                  |
| %T               | Timezone setting of this machine                                                                                                   |
| %U               | Seconds since the Unix Epoch (January 1 1970 00:00:00 GMT)                                                                         |
| %w               | Numeric representation of the day of the week                                                                                      |
| %W               | ISO-8601 week number of year, weeks starting on Monday (added in PHP 4.1.0)                                                        |
| %Y               | A full numeric representation of a year; 4 digits                                                                                  |
| %y               | A two digit representation of a year                                                                                               |
| %Z               | The day of the year (starting from 0)                                                                                              |
| %Z               | Timezone offset in seconds. The offset for timezones west of UTC is always negative, and for those east of UTC is always positive. |

## treemenu

The `treemenu` operator can be used to build tree-like menu structures. It expects the following arguments:

| Parameter                 | Type    | Default                                                    | Required | Description                                                                                      |
|---------------------------|---------|------------------------------------------------------------|----------|--------------------------------------------------------------------------------------------------|
| <code>path</code>         | array   | <code>false()</code>                                       | yes      | An array of node IDs which form the path to the current node                                     |
| <code>node_id</code>      | integer | <code>false()</code>                                       | yes      | The current node ID                                                                              |
| <code>class_filter</code> | array   | if <code>false()</code> , actually <code>array(1,8)</code> | no       | An array of class IDs to include, as in the <code>content/list</code> function                   |
| <code>depth_skip</code>   | int     | <code>false()</code>                                       | no       | The start depth in the node tree where you want your menu structure to start from the root level |
| <code>max_level</code>    | int     | <code>false()</code>                                       | no       | The maximum depth your menu should have                                                          |

The returned array consists of sub-arrays containing:

- `$tmpNodeID`: The node ID
- `level`: The depth level relative from the root level
- `url_alias`: The URL alias for this item
- `url`: The normal URL for this item
- `text`: The node name
- `is_selected`: A boolean to determine if the item is on the current node path

The following example illustrates the use for a top-level menu consisting of just one level. Here we don't want the "tree" functionality, but rather the element that is "active" for highlighting.

```
{let menuLevel1=treemenu($module_result.path,
 $module_result.node_id,
 array('folder'),
 0,
 1)}
{* loop over the menus and build the menu structure *}
{section name=Menu loop=$menuLevel1}
 {section show=$Menu:item.is_selected}

 {$Menu:item.text}
 {section-else}

 {$Menu:item.text}
 {/section}
 {delimiter} | {/delimiter}
{/section}
{/let}
```

# Index

## A

- ab**, 335
- access control limitations**, 32
- accessibility standards**, 310
- action extension**, 183
- actions**, 183
- add an extension**, 150
- adding content**
  - create content classes, 37
- admin view, case study**, 261
- administrator functions**, 61
- advanced keywords**, 122
- anonymous user submissions**, 146
- Apache Benchmarking tool**, 335
- Apache virtualhost**, 7
- APC Alternative PHP Cache**, 249
- array matching**, 99
- array operators**, 344-348
- array\_sum operator**, 344
- attribute() function**, 130, 221
- attribute templates**, 82
- authentication methods**
  - external, 227
  - internal, 227
  - Kerberos, 227
  - LDAP, 227
  - Microsoft Windows domain controller, 227
  - single sign-on, 228
- authorization**, 31
- Awstats**, 238

## B

- backup**, 251
- bandwidth**, 236
- begins\_with operator**, 344
- benchmarking**, 335
- benefits of eZ publish**, 311
- blog site package**, 17
- breadcrumb navigation**, 115

## C

- cache administration**, 64
- cache block parameters**
  - expiry, 110
  - keys, 110
- cache blocks**, 333
- cache blocks**, 109
- callback**
  - about, 212
  - creating callback, example, 213-214
  - enabling callback, 212
  - passing variables, 214
  - permissions, 218
  - testing, 218
  - user settings, 217
- cascading templates**, 112
- case**
  - mandatory attributes, 299
- case study 1**
  - archiving, 305
  - client, 253- 254
  - client requirements, 257
  - client training, 301
  - CMS selection, 258
  - content areas, 282
  - content model, 265
  - content model, final, 266
  - content model, original, 266
  - content population, review, 304
  - CRM integration, 305
  - deployment, 301
  - development lifecycle, 303
  - development, display logic, 280
  - development, installation, 278
  - development, navigation, 282
  - development, roles and permissions, 279
  - development, sections, 279
  - development, templates, 281
  - existing site, 255
  - extending the site, 304

- HTML prototype, 274
- interface design, 272
- maintenance, 301
- process flowchart, 256
- project assessment, 302
- project requirements, 255
- sections, admin view, 261
  - sections, links, 263
- sections, miscellaneous, 264
- sections, user view, 259
- site specifications, 259
- support, 301
- template design, 267
- testing, 300
- workflow, need for, 305
- case study 2**
  - client, 311
  - content design, 328-331
  - CSS rules, 328
  - existing design, 311
  - existing structure, 312
  - optimization effects, 336
  - performance, 332
  - planning, 314
  - resources, 337
  - structural changes, 315
  - template design, 316-323
- changing case, 341**
- class templates, 329**
- classes**
  - Categorytype, 221
  - eZContentClass, 128-132
  - eZContentClassAttribute, 134
  - ezContentObject, 135
  - eZContentObject, 128
  - ezContentObjectAttribute, 138
  - eZPersistentObject, 128, 221
  - eZSearch, 146
- coding functions, 163**
- coding operators, 177**
- comment button, adding, 117**
- commenting files, 84**
- comments, 84**
- community sharing**
  - eZ publish contributions, 205
  - public SVN, 205
  - SourceForge, 206
- comparing strings, 344**
- configuration file extensions, 10**
- content classes**
  - about, 28, 132
  - attributes, 134
  - create, 132-133
  - how to create, 133
  - identifier, 132
  - states, 132
- content management system, 25**
- content model, 265**
- content module functions, 101-104**
- content object classes, 127**
- content object versions, 29**
- content objects**
  - about, 27-28, 135
  - attributes, 137
  - create, 48, 136
  - editing, 50
  - related content, 53
  - states, 136
  - translations, 51
- content sources, 84**
- content types, case study 1**
  - Article, 269
  - client, 269
  - links, 269
  - news, 269
  - Overview Article, 269
  - publication, 269
  - training course, 269
- content view, 30, 81**
- contentless nodes, 119**
- corporate site package, 17**
- count operator, 357**
- create() function, 221**
- creating a new class, 329**
- creator, showing, 122**
- cron jobs**
  - about, 247
  - runcronjobs.php script, 9
- crontab, 9**
- cross-template variables, 93**
- CSS, 309**
- custom content type, publication**
  - attachment, 270
  - attachment, description, 270



- author, 270
- cost, 270
- date, 270
- description of contents, 270
- publisher, 270
- title, 270
- custom operators, 176**
- custom threaded forum template, 124**
- custom-built CMS, 255**

## D

- data import/export, 187-190**
- data interoperability, 196**
- datatype extension, 152**
- datatype functions**
  - fetchClassAttributeHTTPInput(), 169
  - fixupClassAttributeHTTPInput(), 170
  - metaData(), 172
  - storeClassAttribute(), 170
  - title(), 172
  - validateClassAttributeHTTPInput(), 170
- datatype wizard, 165**
- datatypes, 164**
  - class attributes, working with, 169
  - complex datatypes, 173
  - construction, 167
  - datatype wizard, 165-166
  - default class values, 168
  - default object values, 168
  - implementation, 167
  - initialization, 168
  - object attributes, working with, 171
  - settings, 164
  - storing information, 167
  - templates, 164
  - types of, 38-45
- date tasks, 117**
- dedicated hosting, 239**
- definition() function, 129, 221**
- deploying eZ publish**
  - budget, 239
  - deployment checklist, 249
  - hosting requirements, 5
  - number of visitors, 236
  - reporting, 238
  - security needs, 237
  - system requirements, 236

- time limits, 239
- design folder, 6, 12**
- directories, 207**
- disable a module/function, 36**
- displaying date, 117**
- documentation**
  - access control, 243
  - contact details, 241
  - disaster recovery, 243
  - DNS information, 242
  - extension, 206
  - hardware, 242
  - how to update, 241
  - location, 242
  - operating system, 242
  - patches, 242
  - software, 242
  - TCP/IP information, 242
  - upgrading, 243
- download eZ publish, 5**
- Doxygen, 153**
- dummy nodes, 119**

## E

- e-commerce module extension, category datatype**
  - activation, 219
  - database table, 220
  - database, communication, 221
  - editing the class, 224
  - editing the object, 224
  - templates, 224
  - viewing the object, 224
  - working, 221-223
- e-commerce module extension, WorldPay**
  - create workflows, 208
  - eventtype directory, 209
  - extension directory, 209, 213
  - how it works, 215-216
  - ini settings, 211-217
  - key functions, 214
  - known issues, 212
  - module, creating, 213-214
  - overview, 209
  - PHP source code, 210
  - requirements, 207

- workflow routine, 211
  - WorldPay module, 212
- edit functions, 116**
- edit link, adding, 117**
- edit template variables, 92**
- edit templates, 82**
- ends\_with operator, 344**
- error codes, 128**
- example site**
  - activities calender, 71
  - create basic classes, 69
  - discussion forums, 71
  - document types, 70
  - images, 70
  - members, 77
  - personalization, 72
  - role assignments, 76
  - roles, 74
  - sections setup, 74
  - user groups, 75
  - visitors, 77
- execute() function, 179**
- executing operators, 177**
- expiry, cache blocks, 110**
- explode() function, 222**
- extension development, practices**
  - defining goals, 204
  - development tools, 205
  - documentation, 206
  - sharing, 205
  - software requirements, 204
  - support forums, 204
  - testing, 204
  - timescales, 204
  - upgrades/updates, 206
- extensions**
  - actions, 183
  - add an extension, 150
  - category datatype, 219
  - designing an extension, 203-206
  - documentation, 153
  - e-commerce module, example, 206
  - locate an extension, 151
  - subdirectories, 151
  - template operators, 173
  - third party extensions, 68
  - translations, 184
- extract operator, 343-346**
- extract\_left operator, 343**
- extract\_right operator, 343**
- ezauthor datatype, 38**
- ezbinaryfile datatype, 38**
- ezboolean datatype, 38**
- ezdate datatype, 38**
- ezdatetime datatype, 38**
- ezdesign integration, 281**
- ezemail datatype, 39**
- ezenum datatype, 39**
- ezfloat datatype, 40**
- eZHTTPTool eZ library routine, 214**
- ezidentifier datatype, 40**
- ezimage datatype, 40**
- ezimage integration, 281**
- ezinisetting datatype, 41**
- ezinteger datatype, 40**
- ezisbn datatype, 41**
- ezkeyword datatype, 41**
- ezlink attribute, 62**
- ezmatrix datatype, 42**
- ezmedia datatype, 42**
- eZModuleOperationInfo() function, 140**
- ezobjectrelation datatype, 42, 54**
- ezobjectrelationlist datatype, 43, 54**
- eZOperationHandler() function, 140**
- ezoption datatype, 43**
- ezpackage datatype, 43**
- eZPersistentObject class, functions**
  - fetching data, 131
  - hasAttribute(), 131
  - other functions, 132
  - storing data, 131
- ezpreference system, 93**
- ezprice datatype, 43**
- eZSearchEngine plug-in, 146**
  - searching, 65
  - available filters, 148
  - description, 146
  - module views, 148
  - setup, 147
- ezselection datatype, 44, 165**
- ezstring datatype, 44**
- ezsubtreesubscription datatype, 44**
- eZ systems, 1**
- eztext datatype, 44**

- eztime datatype, 38**
- ezurl datatype, 45**
- ezuser datatype, 45**
- ezxmltext attribute, 62**
- ezxmltext datatype**
  - about, 45
  - custom tags, 48
  - headings, 46
  - hyperlinks, 47
  - lists, 46
  - objects, 47
  - tables, 47
  - text emphasis, 47
  - unformatted text, 47

## F

- fetch() function, 104, 221**
- fetchByAttribute() function, 223**
- fetchByCategoryAndAttribute() function, 221**
- fetchClassAttributeHTTPInput() function, 169**
- fetch parameters, 105-106**
- fetching a single node, 104**
- fetching node lists, 105**
- fetching the current user, 108**
- filtering classes, 106**
- folder object, publishing, 197**
- folders, 6**
- forum site package, 17**
- forum template, custom, 124**
- functions**
  - attribute(), 130, 221
  - attributes(), 131
  - create(), 221
  - definition(), 129, 221
  - execute(), 179
  - explode(), 222
  - eZModuleOperationInfo(), 140
  - eZOperationHandler(), 140
  - fetch(), 104, 221
  - fetchByAttribute(), 223
  - fetchByCategoryAndAttribute(), 221
  - fetchClassAttributeHTTPInput(), 169
  - hasAccessTo(), 127
  - hasAttribute(), 131
  - implode(), 223
  - list(), 105
  - redirect(), 161

- remove(), 221
- setAttribute(), 130
- store(), 222
- storeObjectAttribute(), 222
- strtoupper(), 174
- tree(), 105
- upcase(), 174
- validateClassAttributeHTTPInput(), 170
- version\_list(), 108
- view(), 31

## G

- gallery site package, 17**
- GD, 8, 11**
- Google integration, 230-232**

## H

- hasAccessTo() function, 127**
- hasAttribute() function, 131**
- hosting, choosing**
  - time limits, 239
- hostname site access type, 18**
- htaccess, 7, 20**

## I

- image operator, 355**
- image optimization,**
  - about, 8, 354
  - GD, 8, 247
  - image operator, 355
  - ImageMagick, 8, 247
- image.ini, 10**
- imagefile operator, 356**
- ImageMagick, 8, 11**
- implode() function, 223**
- implode operator, 346**
- information array, 284**
- information collector**
  - about, 48
  - policies, 146
  - setting up, 145
  - types, 145
- ini files**
  - browse.ini, 54
  - collect.ini, 145
  - content.ini, 29

- content.ini.append.php, 183, 217
- design.ini.append, 217
- i18n.ini, 10
- image.ini, 10
- layout.ini.append, 217
- ldap.ini, 200
- module.ini.append, 217
- override.ini, 10
- override.ini.append.php, 112
- site.ini, 10
- template.ini, 173
- workflow.ini.append, 217
- worldpay.ini.append.php, 217
- initialization files, example, 283**
- initializing operators, 177**
- installation**
  - Apache setup, 244
  - Apache virtual host settings, 7
  - database setup, 246
  - e-mail setup, 248
  - Image settings, 8
  - Initialize the database, 6
  - linux environment, 243
  - PHP accelerators, 248
  - PHP setup, 245
  - Setup Wizard, 10-22
  - troubleshooting, 22
  - Unpack files, 5
- integration, 225**
  - authentication, 227
  - bridging external applications, 225
  - communicate with Google, 230
  - identification, 227
  - modifying existing code, 233
  - SOAP, 230
  - strategies, 226
- interface design, case study**
  - HTML prototype, 274
  - visual design, 272
- intranet site package, 17**
- ionCube accelerator, 249**

## J

- joining strings, 343**

## K

- kernel**
  - classes, 5
  - folder, 10
  - module functions, 101
  - operators, 359
- keys, cache blocks, 110**
- keywords, advanced, 122**
- keywords, listing, 122**

## L

- l10n operator, 350**
- l18n.ini, 10**
- lib folder, 6**
- libraries**
  - lib/ezsoap/classes/ezsoapclient.php, 230
  - lib/ezsoap/classes/ezsoaprequest.php, 230
- limitation checks, 127**
- links, case study, 263**
- list() function, 105**
- listing keywords, 122**
- logical operators, 351-353**
- login handlers**
  - LDAP, 199
  - text file, 201

## M

- mathematical functions, 348-349**
- media tab, 66**
- menus**
  - breadcrumb navigation, 115
  - top level, 114
  - tree, 115
- metadata standards, 229**
- missing users sections, 67**
- mod\_rewrite, 7**
- modifier, showing, 122**
- modularization, 82**
- module extension, 152**
- module functions, 162**
  - coding a function, 163
  - registering a function, 162
- modules**
  - about, 153
  - coding, 159

- defining a module, 154
- module functions, 162
- processing a template, 160
- reading input, 159
- redirecting a module, 161
- registering a module, 153
- returning information, 159
- using templates, 160
- view actions, 156
- view parameters, 155
- view permissions, 155

**multi-language support, 16**

## N

**nameless sections, 100**

**navigation menus, 114**

**newDatatype datatype, 165, 167**

**news site package, 17**

**nice urls, 81**

**nmap, 251**

**node tree, 26**

**nodes**

- counting, 107
- fetching, 104
- filtering, 106
- sorting, 107

**nodes without content, 119**

**nodes, counting, 107**

**normal post variables, 158**

**notification events, 142, 185**

- collaborations, 187
- event handlers, 143
- notification system, 68
- publish operation, 143
- statuses, 145
- subtree notification, 143

**notification handlers, 144**

## O

**object model, 5**

**objects**

- attributes, 128
- persistent object model, 128
- removing/restoring, 61

**operation, 139**

- definition, 139

- processing, 140
- statuses, 140

**OpenFITS plug-in, 146**

**operators**

- choose, 359
- coding, 177
- contains, 359
- count, 357
- count\_chars, 344
- creating, 176
- currentdate, 359
- executing, 177
- extract, 343
- ezini, 358
- ezroot, 359
- ezurl, 359
- image, 355
- imagefile, 356
- initializing, 177
- kernel, 359
- localization, 349
- logical operators, 351
- registering, 176
- shorten, 118
- striptags, 118
- translation operators, 349
- treemenu, 361
- type checking operators, 353-354
- wrap, 342

**override folder, 9**

**override.ini, 10**

**override.ini.append.php, syntax, 112**

**overrides, 329**

**overriding templates, 112**

## P

**packages, 68**

**pad operators, 343**

**page layout template, case study 1**

- determine top node ID of section, 285
- development lifecycle, 281
- information array, 284
- initialization files, 283
- initialize variables, 283
- primary navigation, 286
- secondary navigation, 288
- secondary navigation, code, 290

- variables, assign values, 285
- page layout view, 30**
- PDF export, 67**
- performance, improving, 109**
- permissions,**
  - about, 5, 57
  - folder permissions, 12
  - identifying users, 127
  - setting up, 126
- persistent object model, 128**
- personal tab, 68**
- PHP accelerators**
  - about, 234, 248
  - Alternative PHP Cache, 335
  - IonCube PHP Accelerator, 335
  - Turck MMCache, 335
- phpMyAdmin, 7**
- plain site package, 17**
- plug-ins, 146**
- policies, 5, 31**
- policy limitations, types, 126**
- port scanning, 251**
- port site access type, 18**
- ports, 251**
- post variables, 156**
- post\_action\_parameters variable, 157**
- post\_actions variable, 157**
- PostgreSQL, 14**
- pre\_publish trigger, 140**
- problems**
  - memory limits, 22
  - PHP running as CGI, 23
- processing an operation, 140**
- project requirements. *See* typical project requirements**
- Public Key Infrastructure (PKI), 238**
- publication content type, *See* custom content type, publication**
- publish operation, 143-144**
- publishing date, showing, 122**

## Q

Qt Linguist, 350

## R

RAD, 68

- redirect() function, 161**
- redirecting modules, 161**
- registering functions, 162**
- registering operators, 176**
- related content, 53**
- related objects panel, 53**
- relationships**
  - content objects and notification system, 143
  - eZContentObject class and tables, 135
  - objects and eZSearchEngine plug-in, 147
  - objects, roles, permissions, and limitations, 126
  - workflows and triggers, 139
- remove button, adding, 117**
- remove() function, 221**
- restrictions. *See* permissions**
- reusing template code, 82**
- role system, 35**
- roles, 5, 31**
- RSS, 62, 193,**
- RSS admin page, 194**
- RSS import, 195**

## S

- sample site, create, 69**
- scalar matching, 99**
- searching, *See* eZSearchEngine plug-in**
- section function, 94-98**
- section parameters, 94**
- sections, 27, 66, 88**
- Secure Socket Layer (SSL), 238**
- security**
  - htaccess, 20
  - levels, 237
  - requirements, 237
  - virtualhost mode, 7, 20
- security, requirements, 237**
- semantically correct markup, 309**
- sendmail, 13**
- seperation, content and application logic, 30**
- server, choosing, 240**
- setAttribute() function, 130**
- settings folder, 6, 9, 12**
- Setup Wizard, 10-22**
- shared hosting**
  - time limits, 239
- Shop tab, 69**

- shorten**, 343
- shorten operator**, 118
- single sign-on**, 228
- site access types**
  - Hostname, 18
  - Port, 18
  - URL, 18
- site packages**, 17
- site statistics**
  - reporting, 238
- site structure**
  - content class attributes, 28
  - content classes, 28
  - content object, 28
  - node tree, 26
  - sections, 27
- site.ini**, 10
- siteaccess folder**, 9
- SOAP**, 149, 187-197, 225-233
  - example, 188
  - required functions, 190
  - XML client request, 192
  - XML server response, 192
- SOAP example**, 230-232
- standards. See accessibility standards**
  - accessibility, 310
  - Section 508 guidelines, 310
  - web, 307
  - Web Accessibility Initiative, 310
- standards, advantages**, 310
- store() function**, 222
- storeObjectAttribute() function**, 222
- string conversion, automatic**, 119
- string manipulation**
  - automatic linking, 119
  - comparing strings, 343
  - counting strings, 344
  - extracting strings, 343
  - joining strings, 343
  - limiting text output, 118
  - matching, 341
- string operators**, 339-341
- striptags operator**, 118
- structuring content**, 26
- supported systems**, 243
- SVN code repository**, 150
- switch function**, 98

**system information**, 65

## T

- template compiling**, 333
- template functions**
  - {append-block}, 86
  - {default}, 85
  - {let}, 85
  - {section}, 94-97
  - {set}, 85
  - {set-block}, 100
  - {switch}, 98
- template operator wizard**, 174
- template operators**, 173
- template operators, adding your own**, 111
- template output**
  - controlling flow, 94
- template override system**, 111
- template parameters**
  - match, 113
  - matchfile, 113
  - source, 113
  - subdir, 113
- template tasks**, 114
- templates**, 59, 319-323
- templates, edit functionality**, 116
- testing, case study 1**
  - content population, 301
  - functional testing, 300
  - implementing, 300
  - requirements, 300
  - specifications, 300
- text file login**, 201
- translation extension**, 184
  - overriding, 185
  - URL translation, 61
- tree() function**, 105
- tree menus**, 115
- triggers**, 54, 138
  - return statuses, 141
- troubleshooting, see problems**
- Turck MMCache**, 249
- type checking operators**, 353-354
- typical project requirements**, 257-258

## U

- upcase() function**, 174
- URL site access type**, 18
- URL translation**, 61
- user groups, show**, 120
- user panel, creating**, 119
- user preferences function**, 120
- user roles, show**, 120
- user view, case study**, 259

## V

- validateClassAttributeHTTPInput() function**, 170
- var folder**, 6, 12
- var parameter, section function**, 101
- variable scope**, 85-86
- variables**
  - \$attribute, 173
  - \$class\_attribute, 172
  - \$FunctionList, 154
  - \$Module, 154
  - \$module\_result, 89, 159
  - \$module\_result.content\_info, 89
  - \$node, 90
  - \$node.object, 92
  - \$ViewList, 154
  - changing variables outside a loop, 100
  - content view variables, 90
  - definition, 85
  - edit template variables, 92
  - ini variables, 358
  - namespaces, 88, 99
  - pagelayout template variables, 89
  - predefined variables, 88
  - setting and modifying, 85
  - type creators, 87
  - types, 87
- version history audit trail, showing**, 121
- version information, displaying**, 108
- version\_list() function**, 108
- version, content objects**, 29
- view() function**, 31
- view actions**, 156
- view caching**, 333
- view navigation**, 158
- view parameters, types**, 156

- view templates**, 81
- virtualhost**, 7, 20

## W

- Web Accessibility Initiative**, 310
- web standards**, 307
  - CSS, 309
  - XHTML, 308
- Webalizer**, 238
- webshop site package**, 17
- wireframes**, 267
- wizards**
  - datatype wizard, 165
  - template operator wizard, 174
- workflow**, 54, 138
  - operation, 140
  - parent class, 179
  - settings, 178
  - triggers, 138, 181
- workflow events**, 55
- workflow eventtype extension**, 152
- WorldPay extension. See e-commerce module extension, WorldPay**
- wrap operator**, 342

## X

- XHTML**, 307-334
- XHTML compliance**, 307
- XML tags**, 47-48

## Z

- Zend performance Suite**, 249