Balazs Halasy

eZ publish basics

eZ publish basics

by Balazs Halasy

Printed in Norway.

Managing Editor: Jennifer Zickerman
Technical Editors: Zak Greant, Vidar Langseid, Jan Kudlicka
Copy Editors: Mandy Moore, Jennifer Zickerman
Cover Design: Edison Reklamebyrå
Printing: Erik Tanche Nilssen AS

International Standard Book Number:
ISBN-10: 82-92797-00-9
ISBN-13: 978-82-92797-00-9

First printing: June, 2006

eZ press, a division of eZ systems AS
Kverndalsgate 8, Postboks 253, N-3701, Skien, Norway

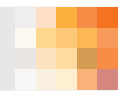
Phone: + 47 35 58 70 20
Fax: + 47 35 58 70 21
Email: info@ez.no

http://www.ez.no

Table of Contents

List of Figures

Preface



eZ publish is the leading Open Source Enterprise Content Management System. It is used to build powerful, flexible web solutions that enable people to share their information. With more than 30,000 downloads per month (to a total of 1.6 million) and 150,000 installations, eZ publish is trusted by enterprises and organizations around the world.

This book, while based on the eZ publish documentation, has been expanded and organized to suit a wider variety of needs. eZ publish course attendees, customers, partners, community members and even employees of eZ systems were looking for something that could be used in different settings. This book attempts to satisfy this group of needs by combining the most popular parts of the online documentation with additional material that shows how to solve everyday tasks and create extensions. Overall, it provides a comprehensive introduction to the fundamental concepts and basics of eZ publish.

About eZ publish

eZ publish is a highly flexible and customizable content management system. It can be used to build everything from personal home pages to multinational corporate web solutions with role-based multi-user access, online shopping, discussion forums and other advanced functions. Based on Open Source technologies and principles, it can be easily extended and customized to interact with other solutions. The following list presents a brief overview of the most important features and benefits of eZ publish:

- Dynamic and customizable content structure
- Separation of content and design
- Web-based Administration Interface
- Powerful and flexible template language
- Platform independence (UNIX/Linux, Windows, OS X, etc.)
- Flexible licensing (Open Source and Professional License options)
- Built-in content versioning
- Support for multiple languages, translations and locales
- Built-in search engine
- E-commerce / Webshop functionality
- Built-in plug-in system
- Role-based permission system

- Built-in workflow system
- Based on open and widely used standards (XHTML, XML, PDF, RSS, PHP, SQL, LDAP, WebDAV, SOAP, etc.)

Target audience and usage

This book is written for web developers who want to learn the architecture of eZ publish. It is targeted at beginners who have little or no experience with the system. It attempts to ease people through the learning curve by providing a clear and well-structured overview of the system. It should answer many of the questions that arise when people start working with eZ publish.

Although it is written by a developer for other developers, the material is presented in a light, comprehensible fashion. Only when necessary does the book go deep into details. Instead, it attempts to provide a broad understanding of how eZ publish works.

The book can be used in a self-study environment or as a textbook for the eZ publish technical course. It can also be used while preparing for the eZ publish web developer certification. It is compatible with eZ publish versions 3.5 to 3.8.

Contents

As mentioned above, this book can be used for multiple purposes. Thus, the different chapters are loosely coupled. While it is not necessary to read the book from start to end, it is recommended that you at least review *the Concepts and basics* chapter before proceeding with other chapters. *Concepts and basics* provides fundamental information about the system architecture, its mechanisms for managing content, configuration files, permissions and so on.

The chapters are organized as follows:

Chapter 1, *Installation*, provides an overview of the installation requirements and describes the installation and configuration process. In addition, it covers miscellaneous issues that are related to the installation process.

Chapter 2, *Concepts and basics*, introduces the fundamental concepts of the system. Developers who are new to eZ publish should read this chapter to learn the architecture and structure of the system. This chapter is more abstract than technical; it is meant to teach broad concepts rather than explain specific features.

Chapter 3, *Templates*, explains the eZ publish template system used for content presentation and interaction. It describes both the template language and the way the system handles the template files. It also covers the template override system.

Chapter 4, *Common solutions*, contains a collection of common solutions, tips and tricks that are useful for a new eZ publish developer. Without delving into too much detail, this chapter contains the information a typical developer needs to create basic but feature-rich eZ publish sites.

Chapter 5, *Extensions*, provides an introduction to extending eZ publish. It is targeted at developers who want to add custom functionality to the system. Because extensions are written in the PHP programming language, some familiarity with programming concepts and practices is

required. After reading this chapter, a moderately experienced programmer will understand the potential of the extension system and should be able to create eZ publish extensions.

The appendix contains a quick reference to the built-in datatypes, a list of the built-in modules and a description of the tags supported by the XML block datatype.

The glossary contains an description of the acronyms used throughout the book.

Typographical conventions

- Code samples, functions, variable names, etc. are printed in `monospace font`.
- Filenames and paths are printed in *`monospace italic font`*.
- Commands are printed in **`monospace bold font`**.
- Elements of graphical user interfaces (such as buttons and field labels) are printed in **bold font**.

About eZ systems

eZ systems is the creator of eZ publish, the leading Open Source Enterprise Content Management System. Based on a philosophy of *openness and information sharing*, eZ systems combines enterprise software with open source code and total product responsibility.

eZ systems was established in 1999. As of mid-2006, eZ systems employed approximately 70 people from 18 different countries, with offices in Norway, Ukraine, Germany, France and Canada. The eZ crew is committed to delivering high-quality software and professional services that contribute to the success of our partners, customers and community.

Contact eZ systems

Please visit our website at http://www.ez.no. Send questions, suggestions and comments to info@ez.no.

All books in the eZ press series are available online at `http:// www.ez.no/ezpress`.

Acknowledgments

It is not until you undertake a book project that you realize how much effort it demands and how much you rely on other people's feedback, contributions, encouragement and patience. Numerous individuals helped me along the way. Without them, this book would have never become a reality. I want to give a big thanks to you all.

I would like to start by thanking our chief software engineer, Jan Borsodi, for providing detailed explanations and hints. Even though he is constantly overloaded with a wide range of difficult tasks, he was always patient, polite and helpful.

A big thanks to Thomas Hellstrøm for translating my sometimes chaotic hand-drawn sketches into cool illustrations, and for tolerating my demanding personality. Thomas was also one of the people who worked on the documentation site and on the integration with `http:// www.ez.no`.

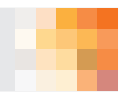
Thanks to Frederik Holljen who helped and supported me when we started the documentation project. Thanks also to Kristian Hole, who in addition provided valuable feedback from various workshops and training sessions. Ole Morten Halvorsen, Kåre Køhler Høvik and Jo Henrik Endrerud also deserve thanks for showing me neat tricks. Jan Kudlicka and Vidar Langseid helped a lot by reading through the code samples and finding mistakes. Thanks to Sergiy Pushchin, Dmitry Lakhtyuk and Vadim Savchuck (also known as ”the guys from Ukraine”) who helped with the initial documentation site and provided various pieces of information.

The eZ press team also deserves a big thanks for their hard work. Zak Greant and Jennifer Zickerman did an excellent job in editing the content. Their patience and professionalism encouraged me through to the very end. I also want to thank Terje Gunrell-Kaste, Hansi Von Grenzfeldt, Lukasz Serwatka, Aleksander Farstad and Bård Farstad for helping with project coordination.

I would also like to thank my family and friends for being patient and tolerant during stressful and difficult times. Last but not least, a big thanks to customers, course attendees, partners, community members and the entire eZ crew for making this happen.

Balazs Halasy

Chapter 1. Installation



This chapter explains how to install and deploy eZ publish using the standard installation method. This option is the most commonly used and recommendedway of installing eZ publish. It assumes that eZ publish is being installed in an environment where the necessary prerequisites are already installed and configured. These prerequisites include a web server and a database management system (like MySQL or PostgreSQL). The actual installation process consists of the following steps:

1. Creating and using a database
2. Downloading a compressed eZ publish distribution
3. Unpacking the eZ publish distribution
4. Configuring eZ publish using the web-based Setup Wizard

Once these steps are completed, eZ publish will be ready to use.

Tip

Note that eZ publish supports multiple installation methods. While only the most commonly used approach will be covered in this book, the other installation methods include:

- Normal installation: Provides a mix of convenience and control. This is the most commonly used and recommended installation method.
- Bundled installation: A bundled installation contains eZ publish and the additional software (Apache, PHP, ImageMagick, etc.) required to run the system.
- Manual installation: The manual installation method is for experienced users who want full control. All configuration is done manually.
- Automated installation: This method provides for automation of the installation process. It is for experienced users and system administrators.

Refer to the online documentation for more information about alternate installation methods.

Installation requirements

eZ publish works in conjunction with four other software components:

- Apache web server (version 1.3+ and 2.x as described below)
- PHP scripting engine (version 4.3.4+, not version 5)
- A relational database management system such as MySQL or PostgreSQL.

- An image conversion system (if image scaling or conversion is required)

The first three software packages should be installed before eZ publish is deployed. The image conversion system is optional and is only needed if you are planning to use eZ publish with images. The web server and the PHP scripting engine must run on the same machine as eZ publish. The database server may be run on a different machine.

The installation and setup of the software prerequisites is beyond the scope of this book. Please refer to the documentation for each application for more information.

Apache web server

The Apache web server is the most popular web server in use. It is Free Software / Open Source and can be downloaded from http:// www.apache.org. The latest stable release of Apache 1.3 should be used; at the time of writing, this was version 1.3.33. We recommend that you use the latest version of the 1.3 branch. However, it is possible to use the 2.x series, but it must run in *prefork* mode instead of *threaded* mode. This is because PHP libraries are not thread-safe. Note that Apache 2.x for Windows is only available in threaded mode and thus it should not be used to run eZ publish solutions on Windows-based systems. The 2.2.x versions might cause problems on all systems.

PHP scripting engine

PHP is Free Software / Open Source and can be downloaded from http:// www.php.net. eZ publish is written in the PHP scripting language and requires version 4 of the PHP scripting engine to run. Version 4.3.4 or later of the engine should be used. Additionally, the engine should include the appropriate modules for communicating with the database server that will be used with eZ publish. At the time of writing, PHP 5 does not work correctly with eZ publish and should not be used. However, future releases of eZ publish will be compatible with PHP 5.

PHP memory limit issue

Ensure that the `memory_limit` setting in PHP's *`php.ini`* configuration file is set to 64 MB or greater. While normal operation of eZ publish only requires about 16 MB, some features and operations (for example, the eZ publish Setup Wizard, PDF export and Unicode (UTF-8) support) require 64MB or more.

Database server

eZ publish stores content and data structures in a relational database. This means that a database server has to be available for eZ publish at all times. By default, eZ publish is compatible with the following relational database management systems:

- MySQL (http://www.mysql.com)
- PostgreSQL (http://www.postgresql.org)

The Setup Wizard will automatically detect the database server as long as it is running on the same computer as the web server. If both MySQL and PostgreSQL are present, the Setup Wizard will prompt the user to select one of the databases. If the database server is running on a dif-

ferent computer, then the Setup Wizard will request the user to enter the necessary connection information.

Oracle compatibility

The eZ publish Database Extension makes it possible to use Oracle as a database for eZ publish. This is a commercial extension which can be purchased from eZ systems. Please refer to the following page for more information: http://ez.no/products/database_extension

Image conversion system (optional)

In order to scale, convert or modify images, eZ publish uses an external image conversion software package. Either of the following software packages (both are Free Software / Open Source) can be used:

- GD graphics library (bundled with with PHP version 4.3+)
- ImageMagick (http://www.imagemagick.org)

ImageMagick supports more formats than GD and usually produces better results (better scaling, etc.). The Setup Wizard will automatically detect the available image conversion software.

Installing eZ publish

The prerequisites for a standard installation must be complete before proceeding with the eZ publish installation. Please read the previous section if you're not sure about the requirements. Proceed only if you have access to a Linux/UNIX/Mac OS X or Windows environment with Apache, PHP, MySQL or PostgreSQL already installed and running. This section will guide you through the following steps:

- Setting up a database (MySQL or PostgreSQL)
- Downloading eZ publish
- Unpacking eZ publish
- Starting the Setup Wizard

Setting up a database

Before running the Setup Wizard, you must create the eZ publish database. The following section explains how to set up a MySQL or PostgreSQL database using their respective command-line interfaces.

MySQL

1. Create a database user for eZ publish:

```
$ mysql -u root -p -e "GRANT ALL \
  ON ezpublish_db.* TO \
  ezpublish@localhost \
  IDENTIFIED BY 'secret'"
```

This example creates a MySQL user called ”ezpublish”. The user’s password will be set to ”secret”. The user will have access to all tables of a database called ”ezpublish_db”.

2. (Optional) For security reasons, remove the ezpublish user’s password from the command-line history:

a. First, show the last two items in the shell’s history using the **history** command.

```
$ history 2
```

b. Second, examine the output of the history command. Find the number of the history entry that contains the command used to create the user. Then run **history** again, using the `-d` (delete) option to delete the history entry.

```
$ history -d 198
```

3. Create a database for eZ publish:

```
$ mysql -u ezpublish -p -e \
  ”CREATE DATABASE ezpublish_db”
```

PostgreSQL

1. Become the PostgreSQL super user (normally called ”postgres”):

```
$ su postgres
```

2. Create a PostgreSQL user for eZ publish:

```
$ createuser ezpublish
```

3. Create a database for eZ publish:

```
$ createdb ezpublish_db
```

4. Log out as the PostgreSQL super user:

```
$ logout
```

Downloading eZ publish

The latest stable version of eZ publish can be downloaded from http:// ez.no/download. Windows users should download the *.zip* archive.

Unpacking eZ publish

Use a dearchiving / decompression tool to unpack the eZ publish distribution to a web-served directory (that is, a directory accessible via a web browser). The following example shows usage of the **tar** utility to unpack a *tar.gz* file on a Linux/UNIX/Mac OS X system. It assumes that the **tar** and the **gzip** utilities are available):

```
$ tar xvfz \
  ezpublish-version_number.tar.gz -C \
  web_served_directory
```

Parameter	Description
version_number	The version number of eZ publish that was downloaded
web_served_directory	Full path to a directory that is served by the web server. This can be the path to the document root of the web server, or a personal web directory (usually called *public_html/ or www/*, and located inside a user's home directory).

The extraction utility will unpack eZ publish into a subdirectory called *ezpublish-3.x.x.* Feel free to rename this directory to something intuitive like *"my_site"*.

The Setup Wizard

This section contains a comprehensive guide to the web-based Setup Wizard of eZ publish. The Setup Wizard is designed to ease the initial configuration of the system. It can be started via a web browser when the installation described in the previous sections is completed.

Initiating the Setup Wizard

The Setup Wizard is run by accessing the *index.php* file located in the root of the eZ publish directory using a web browser. It will be automatically run the first time someone accesses the *index.php* file. The following examples assume that we are using a server with the hostname "www.example.com" and that, after unpacking, the eZ publish directory was renamed *"my_site"*.

Document root example

If eZ publish was unpacked into a directory called *my_site* under the document root, the Setup Wizard can be started by browsing to "http:// www.example.com/my_site/index.php".

Home directory example

If eZ publish was unpacked to a web-served directory located inside the home directory of a user with the username "peter", (usually called *"public_ html"*, *"www"*, *"http"*, *"html"* or *"web"*), the Setup Wizard can be started by browsing to "http://www.example.com/~peter/my_site/index.php".

Tip

To determine which directory name is used for a web-served directory, look at the UserDir directive in the Apache *httpd.conf* file.

Running the installation

The Setup Wizard does not store or modify any data before the final step of the setup is completed. If needed, you may safely re-start the Setup Wizard by reloading the URL but omitting the GET string (for example, "http://example.com/index.php?some+data+here" would become "http://example.com/index.php"). The **back** button (located at the bottom of the Setup Wizard interface) can be used to jump back to previous steps in order to modify settings. (Note that the Setup Wizard varies from version to version. In ad-

dition, some steps may be skipped depending on the environment into which eZ publish is being deployed.)

A typical setup cycle consists of 13 steps:

1. Welcome page
2. System check
3. Outgoing e-mail
4. Database choice (optional)
5. Database initialization
6. Language support
7. Site type
8. Site functionality
9. Access method
10. Site details
11. Site security
12. Site registration
13. Finish

Welcome page

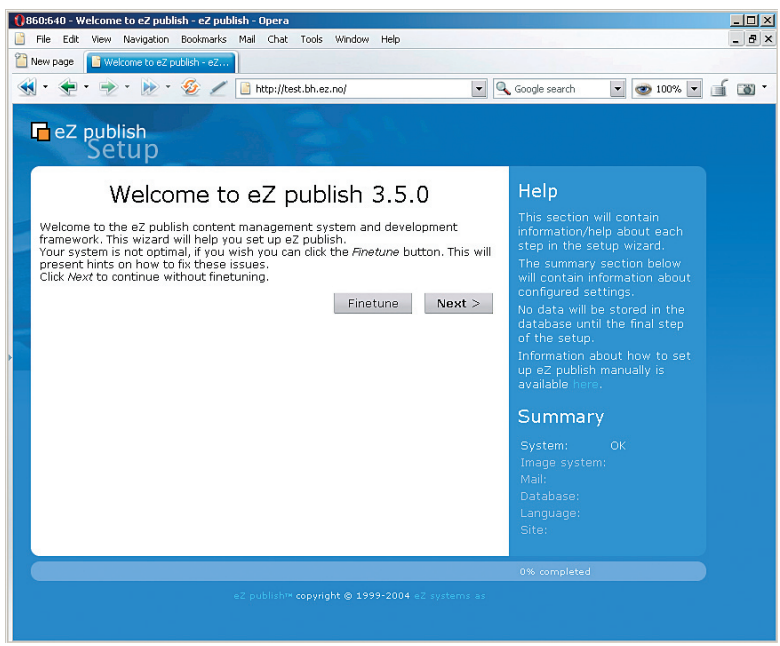


Figure 1.1. Step 01: Welcome page

This is the first page of the Setup Wizard. By clicking **Next**, the Wizard will either proceed to the System Check page (if the installer identifies configuration issues or errors that need to be fixed) or to the Outgoing E-mail page (if everything is okay).

System check

This page usually appears if configuration issues or errors are detected.

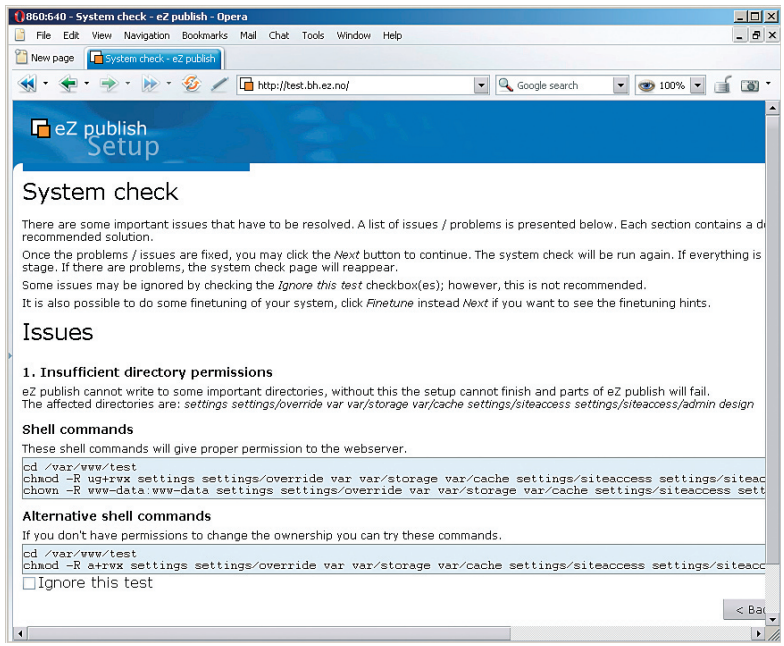


Figure 1.2. Step 02: System check

Issues

One or more problems or errors may be shown on this page. For each problem, a solution is suggested below the description of the problem itself. The Setup Wizard will most likely suggest the execution of various commands (for example, to fix file or directory ownerships, permissions, etc.). These commands must be executed using a system shell. Simply copy the commands from the browser window and paste them into a shell window. The Setup Wizard will run the system check again when the **Next** button is clicked. The System Check page will keep reappearing until all issues have been fixed (or ignored, as described in the next section). When all issues have been fixed, the Setup Wizard will proceed to the next screen.

Warning

If you are unfamiliar with the shell commands suggested by the Setup Wizard, consult a more experienced user for assistance.

Ignoring tests

Some non-fatal errors can be ignored by selecting the checkbox labeled **Ignore this test**. However, we recommend that you fix all errors rather than ignoring them.

Outgoing e-mail

The system uses e-mail to send out messages to users, the site administrator and so on. This page is used to determine how eZ publish delivers outgoing e-mail. There are two options:

- Direct delivery via sendmail (sendmail must be installed and configured on the eZ publish server)
- Indirect delivery using an SMTP server

On Linux/UNIX/Mac OS X, try to use sendmail; use SMTP if sendmail is unavailable. On Windows, use SMTP.

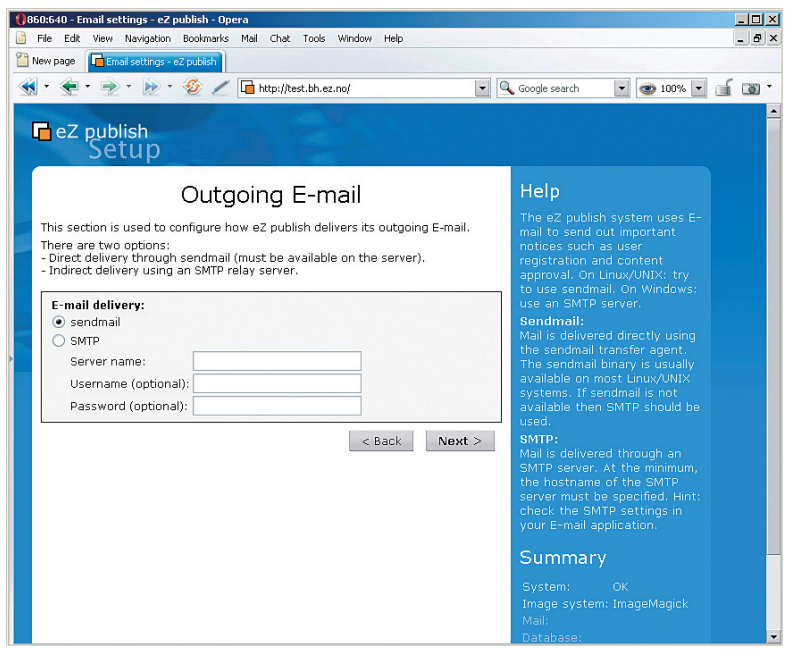


Figure 1.3. Step 03: Outgoing e-mail

Sendmail

E-mail is delivered directly using the sendmail mail transfer agent. The agent must be running on the same host as the web server. The sendmail binary is usually available on most Linux/UNIX/Mac OS X systems. Note that modern Mail Transfer Agents (MTAs) like Qmail and Postfix are able to emulate sendmail. eZ publish will send all outgoing mails to sendmail, which will take care of the actual delivery process. If sendmail or a compatible MTA is not available then SMTP should be used.

SMTP

Mail is delivered via an SMTP server. An SMTP server can be thought of as a local post office. eZ publish will send all outgoing messages to the SMTP server, which will handle the delivery process. At a minimum, the hostname of the SMTP server must be specified. An SMTP server should be used when sendmail is not available.

Database configuration

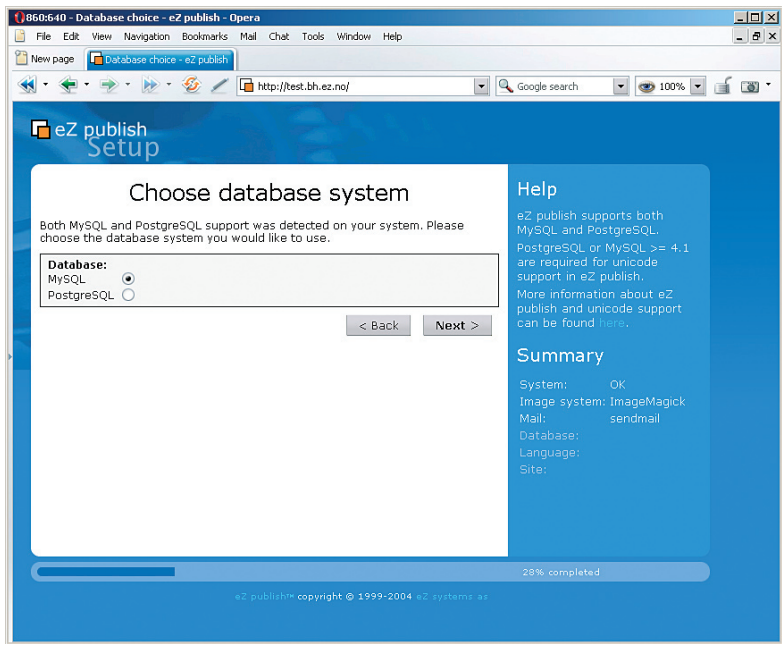


Figure 1.4. Step 04: Database selection

This dialog is used to select the database server. The Setup Wizard will automatically detect the database support configured for the PHP scripting engine. If both MySQL and PostgreSQL are configured, you will be given the option to choose a database. If PHP is configured for only one type of database, eZ publish will automatically use this database and the database choice dialog will not be displayed. If no database is configured, (for example, if the database is running on a different computer), the Setup Wizard will request connection information.

Database initialization

This step provides eZ publish with the required information about the database server. Information about the hostname of the server running the database engine, along with a username and password, must be provided. If MySQL is used, the Setup Wizard will attempt to connect to the database after the **Next** button is pushed. The setup will only continue if it is able to connect to the specified MySQL server with the specified username and password. If PostgreSQL is used, the connection parameters are tested during a later stage of the Setup Wizard.

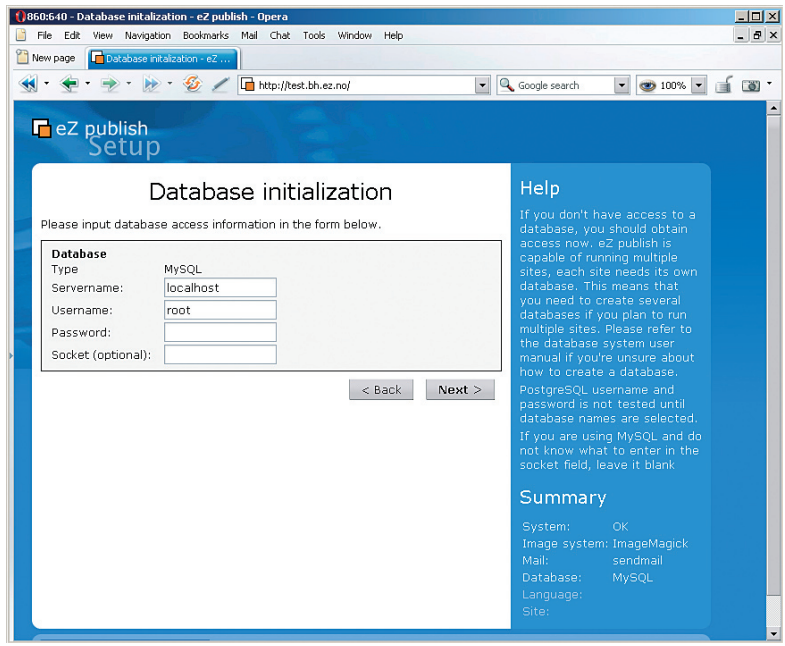


Figure 1.5. Step 05: Database initialization

Language support

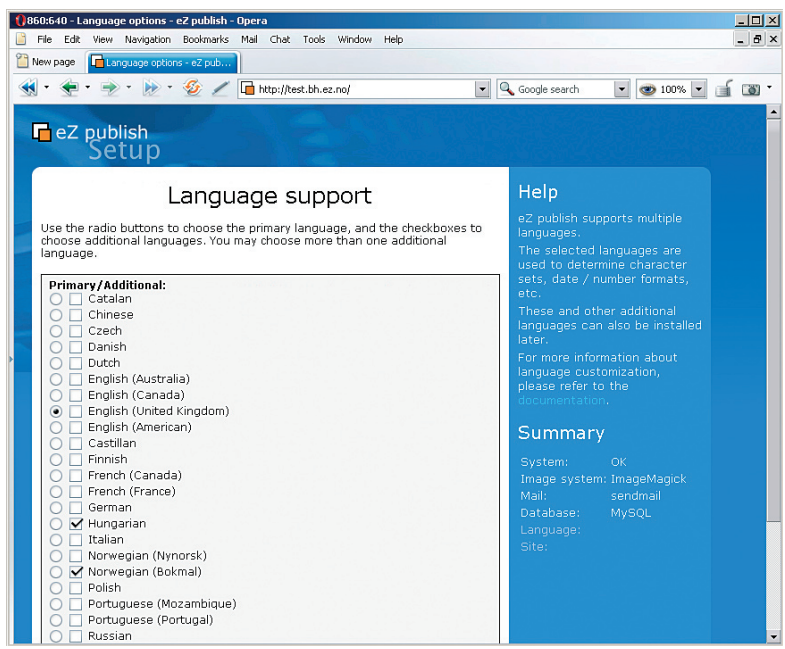


Figure 1.6. Step 06: Language support

This step is used to configure the primary language for the eZ publish installation. You can also specify one or more additional languages. Additional languages enable the translation of site content. For example, additional languages make it possible to have the same news article available in both English and Norwegian. The choice of primary language also affects things like date and number formats, the language used by the Administration Interface and so on. These settings can be changed at a later stage by altering the eZ publish configuration files. Additional languages can be reconfigured at any time (even when the site is up and running) via the Administration Interface. (This step looks different in eZ publish 3.8, as the multilanguage feature was enhanced in that version.) Refer to the *Multiple languages* section of the *Concepts and basics* chapter for more information.

Site type

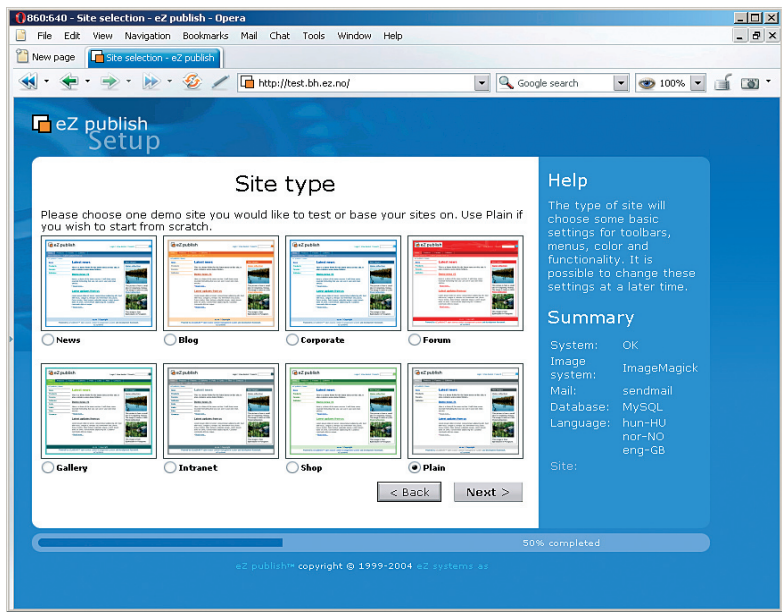


Figure 1.7. Step 07: Site type

This step is used to specify a built-in site type. eZ publish comes with a several basic site types (**News**, **Company**, **Intranet**, **Gallery**, etc.). These examples are mostly for the purpose of demonstration and learning. However, it is possible to choose one example and use it as a basic framework for a custom site. A demo site usually contains some artwork (images), CSS code, content and template files. The **plain** type should be used to set up a custom site. It contains a minimum set of templates and features. (This part of the Wizard looks different in eZ publish 3.8 because of the new package system.)

Site functionality

This step is used to select additional features and demo data for installation. For example, you can select a package that includes contact form functionality. The features in this list are simply additional data structures (content classes) along with display logic (templates) that will be added to the final configuration. These packages can also be installed at a later stage using the *packages* section of the Administration Interface.

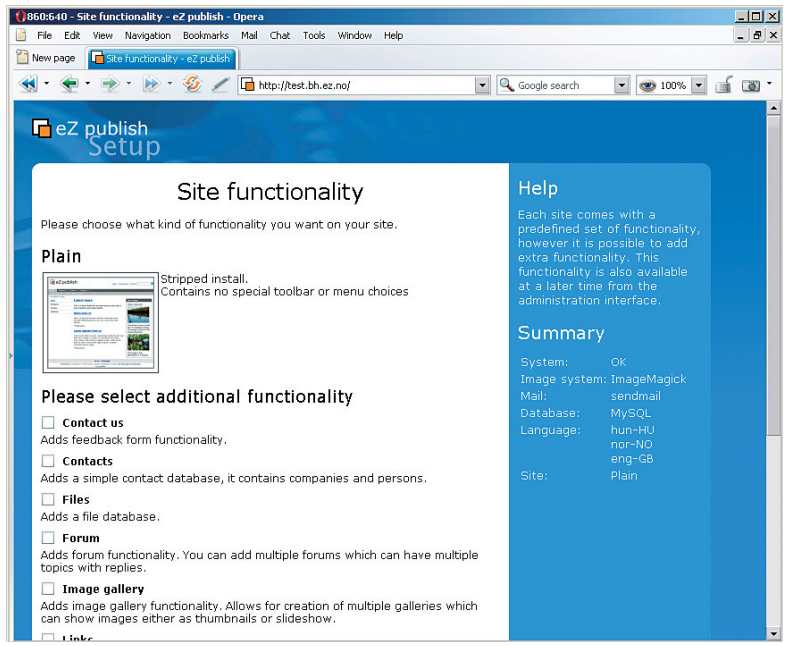


Figure 1.8. Step 08: Site functionality

Access method

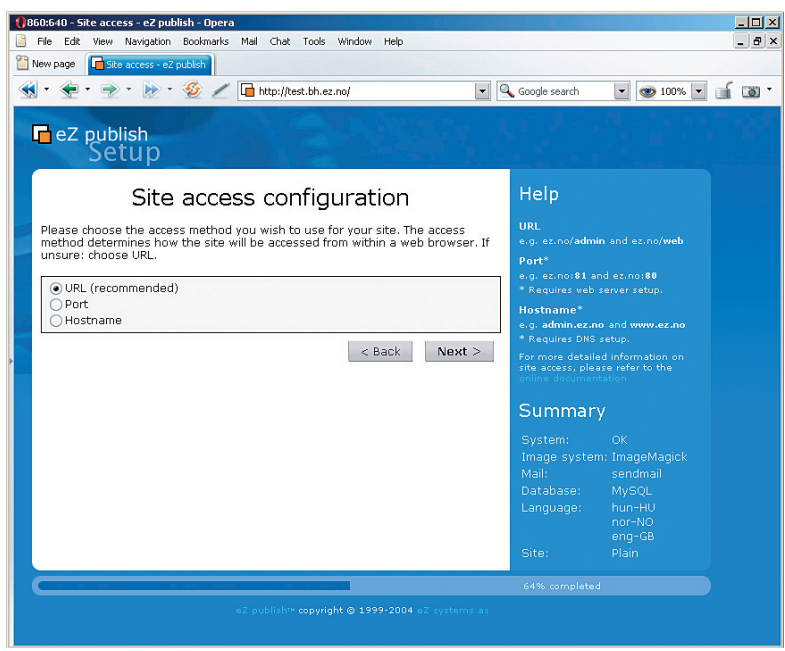


Figure 1.9. Step 09: Access method

This step is used to configure the access method that should be used when eZ publish receives a request. There are three options:

- URL
- Port
- Hostname

URL

When the URL access method is used, eZ publish selects the site that should be accessed based on the contents of the URL (in particular the part following *index.php*). This is the default and most generic option. It doesn't require any additional configuration. Use this setting when installing eZ publish for the first time.

Port

When the port access method is used, eZ publish selects the site that should be accessed based on a port number that is specified in the URL. The port number must be appended to the hostname of the web server, for example: `"http://www.example.com:81/index.php"`. This option requires additional web server and firewall configuration.

Hostname

When this access method is used, each site is assigned a unique hostname. For example, `"www.example.com"` and `"admin.example.com"` can be assigned to the public and the Administration Interface respectively. This option requires additional web and DNS server configuration.

Site details

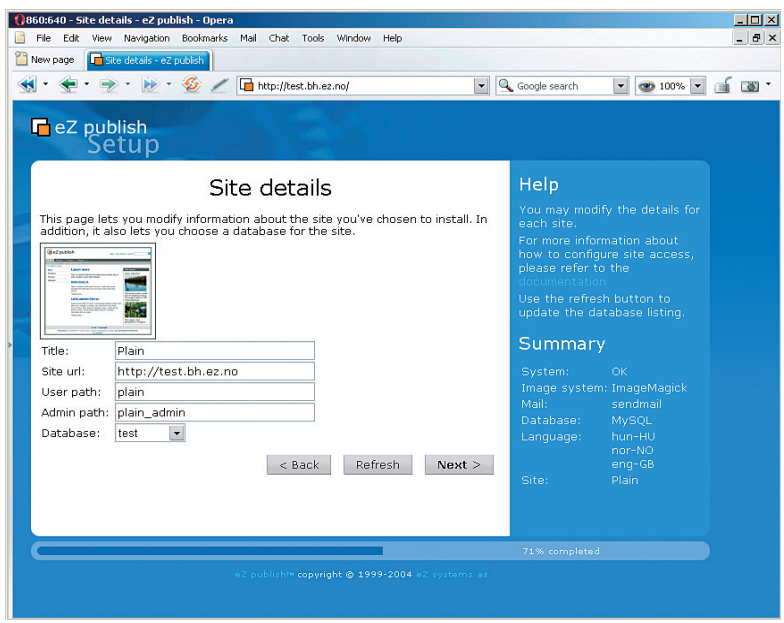


Figure 1.10. Step 10: Site details

This step is used to modify settings related to the site that is being installed. The available databases will be displayed in the database dropdown menu. The **Refresh** button can be used to update the list (if a database is being created while the Setup Wizard is running). If the selected database already contains data, the Site Details page will reappear and ask 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

Use the last option if another database has been chosen. If the chosen database already contains data that can be deleted then use the **"Remove existing data"** option.

Site security

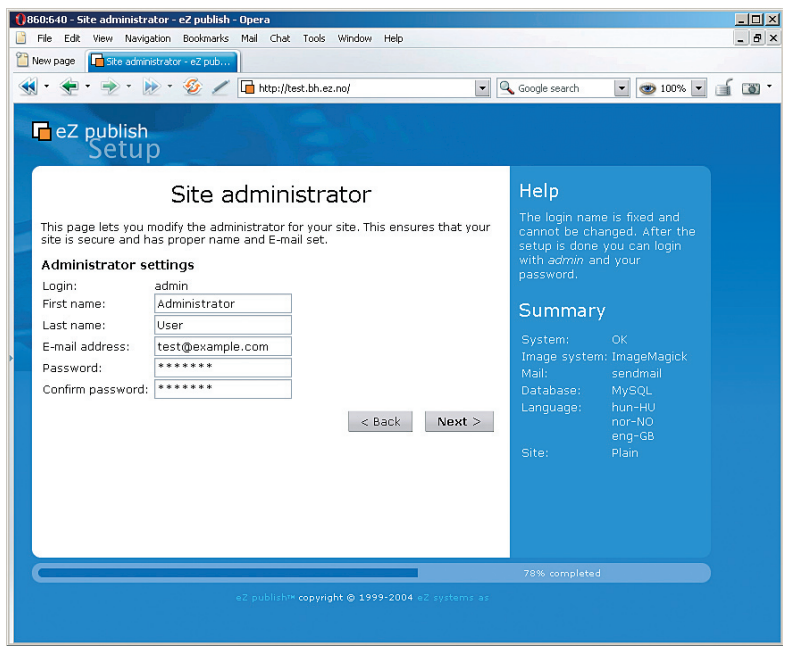


Figure 1.11. Step 11: Site security

This step suggests some basic modifications that should be carried out to enhance the security of the site being installed. The suggested security improvements help protect the configuration files from unwanted access in a non-virtual host environment. These precautions are only necessary if you are configuring a site for public use.

Site registration

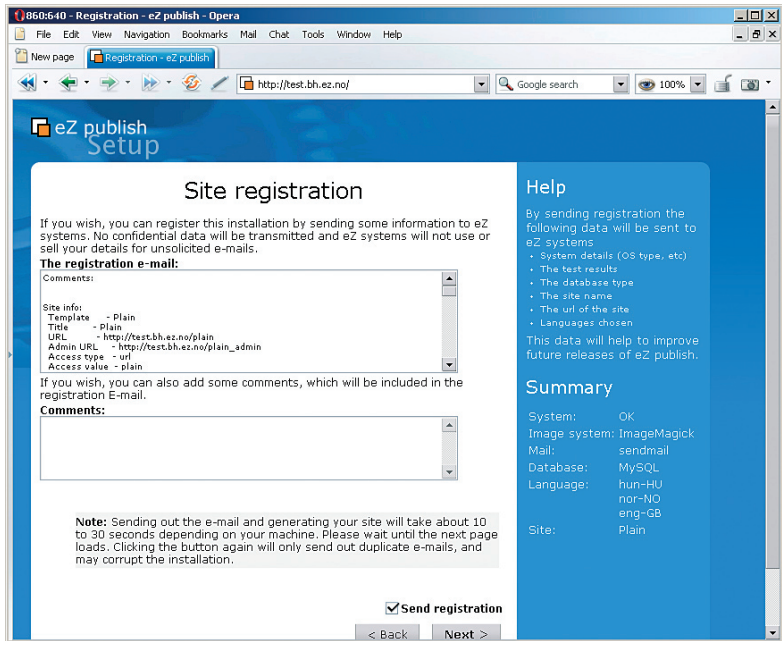


Figure 1.12. Step 12: Site registration

This step allows you to control whether the Setup Wizard should send an email to eZ systems or not. The information in the e-mail will be used internally for statistics and for improving eZ publish. No confidential data will be transmitted and eZ systems will not misuse or sell these details. The following information will be sent:

- System details (OS type, etc)
- Test results
- Type of database being used
- Site name
- Site URL
- Which languages were chosen for the site

Finished

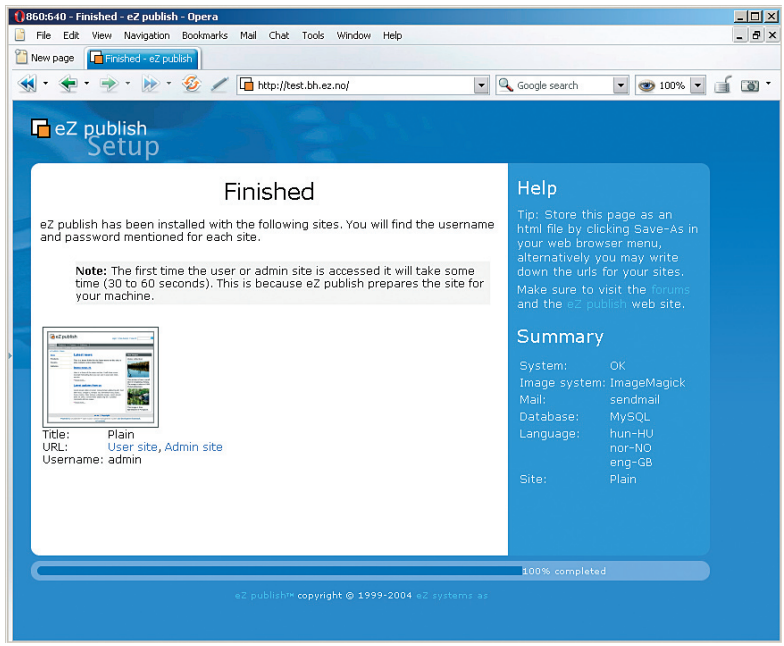
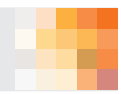


Figure 1.13. Step 13: Finished

The Setup Wizard has finished and eZ publish is ready for use. Use the links to access the various interfaces (public site, Administration Interface, etc.).

Chapter 2. Concepts and basics



This chapter introduces the fundamental concepts of eZ publish. Developers who are new to eZ publish should read this chapter to learn the architecture and structure of the system. This chapter is more abstract than technical; it is meant to teach broad concepts rather than to explain specific functionality. Developers unfamiliar with eZ publish will be able to gain a broad understanding of:

- The architecture of eZ publish
- The directory structure of an eZ publish installation
- The concepts and benefits of separating content from presentation
- How eZ publish stores and manages content
- How custom design is implemented in eZ publish
- The eZ publish configuration system
- How multiple sites can be managed using a single eZ publish installation
- The concept of modules and views
- How eZ publish handles URLs
- The structure of the workflow system
- How access rights and permissions are managed
- How the Webshop works

The architecture of eZ publish

This section describes the internal structure of eZ publish by presenting a brief overview of its software layers. eZ publish is a complex, object-oriented application written in PHP. The system consists of three major parts:

- Libraries
- Kernel
- Modules

The following illustration shows a simplified high-level conceptual view of the relationship between the main eZ publish components.

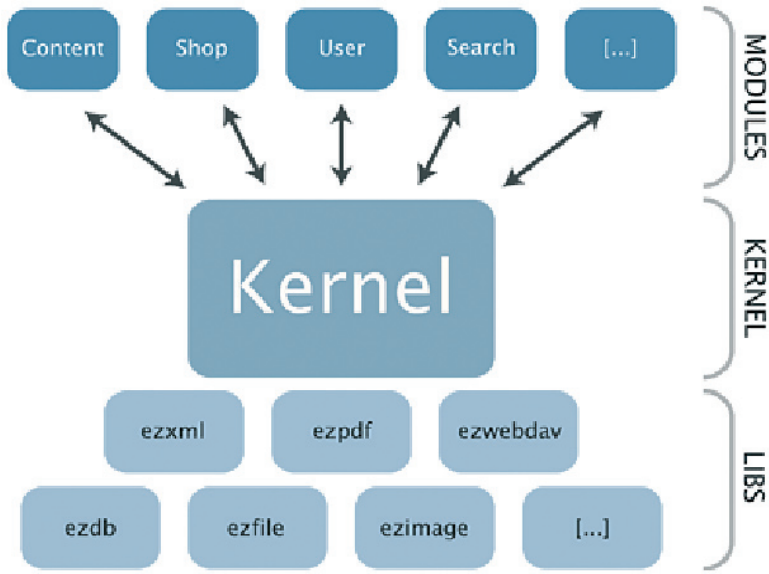


Figure 2.1. Libraries, kernel and modules

The libraries

The libraries contain the reusable general-purpose PHP classes that are the main building blocks of eZ publish functionality. While the libraries are independent of the eZ publish kernel, some are inseparable from other libraries. (General-purpose PHP libraries are located in the `lib/` subdirectory beneath the root directory of an eZ publish installation.)

The kernel

The kernel is the core of eZ publish. It manages low-level and common functionality, such as content handling, content versioning, access control, workflows, etc. The kernel consists of various modules that build upon and make use of the general-purpose libraries. In addition, the kernel contains a collection of PHP classes that are used by the modules.

The modules

The eZ publish *modules* provide HTTP interfaces for web-based interaction with the system. While some modules are an interface to kernel functionality, others are more-or-less independent of the kernel. eZ publish modules contain functionality for common end-user tasks. For example, the `content` module provides interfaces for using a web browser to manage content. The *Modules* section of the appendix contains a complete list and a short description of the available modules. A module can be broken down into the following components:

- Views
- Fetch functions

A *view* is simply a web interface. For example, the `search` view of the `content` module provides a web interface to the built-in search engine. Every eZ publish module provides

at least one view. A `fetch` function extracts data via a module from within a template. For example, the `current_user` fetch function of the user module is used to access information related to the user who is currently logged in. Some modules provide `fetch` functions, some do not.

Directory structure

The eZ publish root directory contains multiple sub-directories. Each subdirectory is dedicated to a specific part of the system and contains a structured collection of related files. The following table gives an overview of the main eZ publish directories.

Directory	Description
bin/	The *`bin/`* directory contains various PHP, Perl and shell scripts. For example, it contains the *`clearcache.php`* script that can be used to clear all eZ publish caches from within a system shell. The scripts are mainly used for manual maintenance.
cronjobs/	The *`cronjobs/`* directory contains miscellaneous scripts used for automated periodic maintenance.
design/	The *`design/`* directory contains all design-related files such as templates, images, stylesheets, etc.
doc/	The *`doc/`* directory contains documentation and change logs.
extension/	The *`extension/`* directory contains eZ publish plugins. Extensions make it possible to create new modules, datatypes, template operators, workflow events and so on.
kernel/	The *`kernel/`* directory contains all the kernel files such as the core kernel classes, modules, views, datatypes, etc. This is where the core of the system resides. These files should not be modified unless you are an expert with eZ publish.
lib/	The *`lib/`* directory contains general-purpose libraries. These libraries are collections of classes that perform various low-level tasks. The kernel uses these libraries.
packages/	The *`packages/`* directory contains the bundled packages (themes, classes, templates, etc.) that can be installed using either the Setup Wizard or the Administration Interface.
settings/	The *`settings/`* directory contains the main configuration files.
share/	The *share/* directory contains static configuration files such as codepages, locale descriptions, translations, icons, etc.
support/	The *`support/`* directory contains the source code for additional applications that can be used to do various advanced tasks. For example, it contains the *`lupdate`* program that can be used to create and maintain the eZ translation files.
update/	The *`update/`* directory contains various scripts that are used when an eZ publish installation is being upgraded.
var/	The *`var/`* directory contains cache files and logs. It also contains content that is stored outside of the database (such as images and files). The size of this directory generally increases as the system is used.

Content and design

This section explains the concepts of content and discusses how design is applied. It is important to understand the difference between content and design, how they interconnect and how the system handles these fundamental elements. One of the keys to building a site that can be easily managed and maintained is knowing how to use the system to clearly separate data from presentation.

Content

In the world of eZ publish, content and design are separate. *Content* is information that is organized and stored by eZ publish. For example, it may be the components of a news article (title, intro, body, images), the properties of a car (make, model, year, color) and so on. In other words, all custom information that is stored for the purpose of later retrieval is referred to as content.

Design

The presentation of content is determined by the design of a site. While content means structured data, *design* refers to the way the data is visually presented. Design includes the things that make up a web interface: HTML, style sheets, images that are not a part of the content, etc.

Templates

An eZ publish template is a custom HTML file that describes how particular types of content should be presented. In addition to standard HTML syntax, it is possible to use eZ publish-specific code to, for example, extract content from the system. The system uses templates as the fundamental unit of site design. For example, a template might dictate that a page should appear with the site's title at the top and with the main content in the middle. When the page is accessed, the content management system places the content into the appropriate locations in the template. The HTML syntax in the built-in templates follows the XHTML 1.0 Transitional specification.

The separation of content and design

While content deals with storing and structuring data, design dictates how the content should be presented. These elements combine to form a complete interface, as illustrated in the following diagram.

The system's ability to handle this distinction is one of the key features of eZ publish. The separation of content and design provides the following benefits:

- Content authors and designers can work separately without conflicts
- Content can be easily published in multiple formats
- Content can be easily transferred and re-purposed
- Global redesigns and changes can be applied via simple modifications

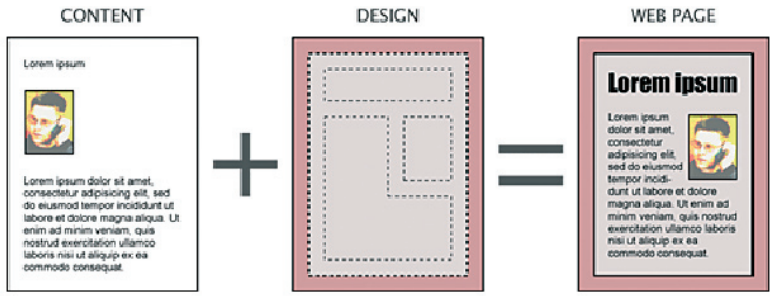


Figure 2.2. Content + Design = Web page

Storage

This section explains where eZ publish stores information that belongs to a specific website. A typical eZ publish site consists of the following elements:

- Content
- Design-related files
- Configuration files

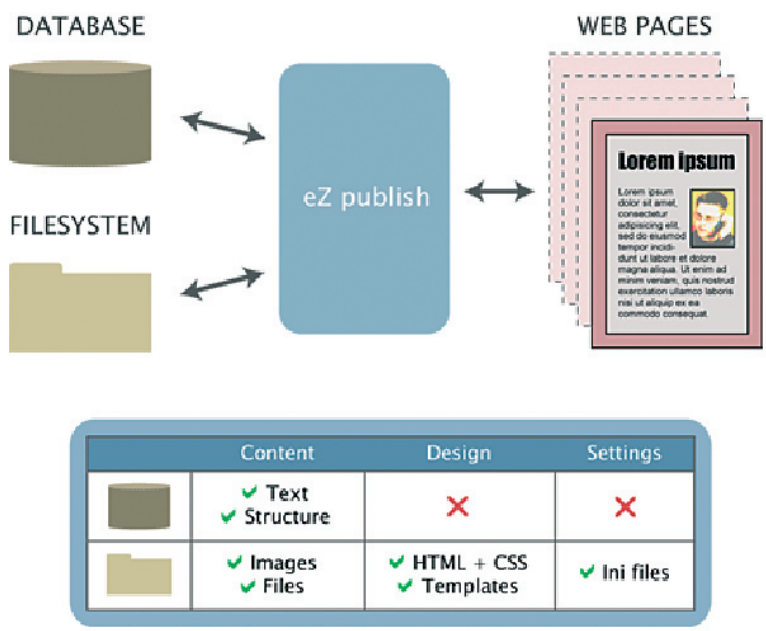


Figure 2.3. Storage overview

Content is structured and stored inside a database. This is true for all content except for images and files, which are stored in the filesystem. The main reason for this is because the filesystem is much faster than the database when it comes to storage and retrieval of large chunks of data.

Storing files in the filesystem allows the web server to serve them directly, without needing to access the database. In addition, this technique makes it easier to use external tools to manipulate, scan and index the contents of the uploaded files. For example, the built-in search engine is capable of using external utilities to index the contents of various file formats (PDF, Word documents, Excel spreadsheets, etc.). (Note that the clustering feature introduced in eZ publish 3.8 makes it possible to store all content in the database.)

Storing the files on the filesystem dramatically decreases the size of the database and makes it easier to manage the data accessed by eZ publish. Everything related to design (template files, CSS files, non-content-specific images, etc.) and configuration settings are also stored on the filesystem. A complete backup of an eZ publish site must contain both a dump of the database and a copy of the necessary files. The following illustration shows an overview of how the system makes use of the database and the filesystem to store the different elements of a site. (Note that as of version 3.8, all content can be stored in the database.)

Content management

This section describes how eZ publish handles content. The role of a content management system is to organize and store content, regardless of type and complexity. The main goal of such a system is to provide a well-structured, automated yet flexible solution, which allows information to be freely distributed and instantly updated across various communication channels (such as the World Wide Web, intranets and miscellaneous front- and back-end systems).

A typical example

Consider this scenario: a university needs to store information about its students. Many content management systems offer static content types that relate to pages of the site. There might, for example, be a ”person” type, consisting of attributes like ”name”, ”birthdate”, ”phone number” and so on. To store information about students, the ”person” type would need to be extended to include critical data like student ID numbers, department and so on. Even though some systems allow for the creation of custom structures, the solution is often complicated and time-consuming, requiring both program modification and manipulation of the database. In addition, subsequent upgrade and modification of the system becomes more complex.

Content management in eZ publish

Unlike many other content management systems, eZ publish does not use a ”one-size-fits-all” approach. Instead of trying to fit data into rigid and predefined structures, the system allows the creation of custom structures using a unique, object-oriented approach. This approach allows for great flexibility and ease of maintenance. For example, using only the Administration Interface, the site developer for the university in our example can easily build custom structures that exactly satisfy the university’s storage needs. This is one of the key features that makes eZ publish a flexible and powerful system.

eZ publish also allows modification of the content structures at runtime. In other words, if the custom student structure in the example needs to be modified, eZ publish will automatically alter it based on the administrator’s actions. Attributes can be easily added, modified or removed while the site is live.

While the ability to create and modify content structures gives eZ publish power and flexibility, the eZ publish distribution also comes with a selection of predefined content structures, thus allowing the developer to choose between the following scenarios:

- Use the built-in structures
- Use modified versions of the built-in structures
- Use only custom structures
- Use a combination of standard, modified and custom structures

An object-oriented content structure

The eZ publish content structure is based on ideas borrowed from objectoriented programming languages like Smalltalk, C++ and Java. Superficially, ”object-oriented” means looking at the world in terms of objects. In real life, people are surrounded by thousands of objects: furniture, cars, pets, humans, etc. Each of these objects has identifying traits. This is conceptually the way that eZ publish defines and manages content.

The system provides a set of fundamental building blocks and mechanisms that combine to provide a flexible content management solution. A data structure is described using a content class. A *content class* is made up of *attributes*. An attribute can be thought of as a field, for example the ”birthdate” field in a structure designed to store information about students. The description of the entire structure would be referred to as the ”student class”. The characteristics of the attributes inside the class are determined by the *datatypes* that were chosen to represent the attributes.

It is important to understand that a content class is just a definition of an arbitrary structure. In other words, the class itself does not store any actual data. Once a content class has been defined, it is possible to create instancesof that class. An instance of a content class is called a content object. Actual content is stored inside objects of different types (for example folders, articles, comments, employees, members, etc.). A content object consists of one or more *versions*. The versioning layer makes it possible to have different versions of the same content. Each version consists of one or more *translations*. The translation layer makes it possible to represent the same version of the same content in multiple languages. A translation consists of *attributes*. The attributes are the final elements in the content structure chain where data is stored.

The content objects are wrapped and organized via *nodes* that are placed inside a tree-like structure. This tree is often referred to as the *node tree*. The tree can in most cases be thought of as the sitemap. The following sections contain comprehensive explanations related to the elements introduced in this section.

Datatypes

A *datatype* describes the type of value that can be stored in a variable. A datatype is the smallest possible entity of storage. It determines how a specific type of information should be validated, stored, retrieved and so on. eZ publish comes with a collection of fundamental datatypes that can be used to build powerful and complex content structures. While these built-in datatypes are sufficient for most scenarios, custom datatypes can also be created. Creating a custom datatype requires PHP programming skills and some knowledge of the eZ publish kernel. The following table provides an overview of the most commonly used built-in datatypes.

Datatype	Description
Text line	Stores a single line of unformatted text
Text block	Stores multiple lines of unformatted text
XML block	Validates and stores multiple lines of formatted text
Integer	Validates and stores a numerical integer value
Float	Validates and stores a numerical floating point value

Refer to the Datatypes section of the appendix for a list of all the built-in datatypes. Additional datatypes, created by the members of the eZ publish community, can be downloaded from http://ez.no/community/contribs/datatypes.

Input validation

As the list above indicates, some datatypes do more than just store data. For example, the XML block datatype supports validation. This means that XML received as input will be validated before it is stored in the database. In other words, the system will only accept and store data if it is a valid XML structure. Input validation is supported by most (but not all) of the built-in datatypes. The validation feature of a datatype cannot be turned on or off. In other words, if a datatype supports validation, it will always try to validate the incoming data and the system will never allow the storage of incorrectly formatted input.

The content class

A *content class* is a definition of a data structure. It does not store any data. A content class is made up of attributes. The characteristics of an attribute are determined by the datatype that is chosen for that specific attribute. By combining different datatypes to represent the attributes, it is possible to create complex data structures. The following illustration shows the anatomy of a content class called `Article`, which defines a data structure for storing news articles. It consists of attributes for storing the article's title, introduction and body.

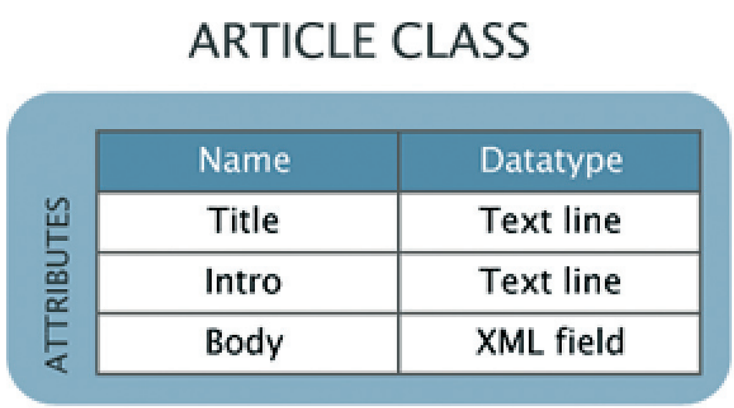


Figure 2.4. Example of a content class

The eZ publish distribution comes with a set of general-purpose classes that are designed for typical web scenarios. For example, the default `image` class defines a structure for managing image files. It consists of attributes for storing the name of the image, the image file, the caption and alternate image text. The built-in classes can be modified to more closely match a specific need. In addition, it is possible to create completely new and custom classes. Content classes are created, modified and removed using the Administration Interface. When a content class is removed, all instances of that class (that is, the objects containing actual data) are also removed from the system. The following screenshot shows the Class Edit interface.

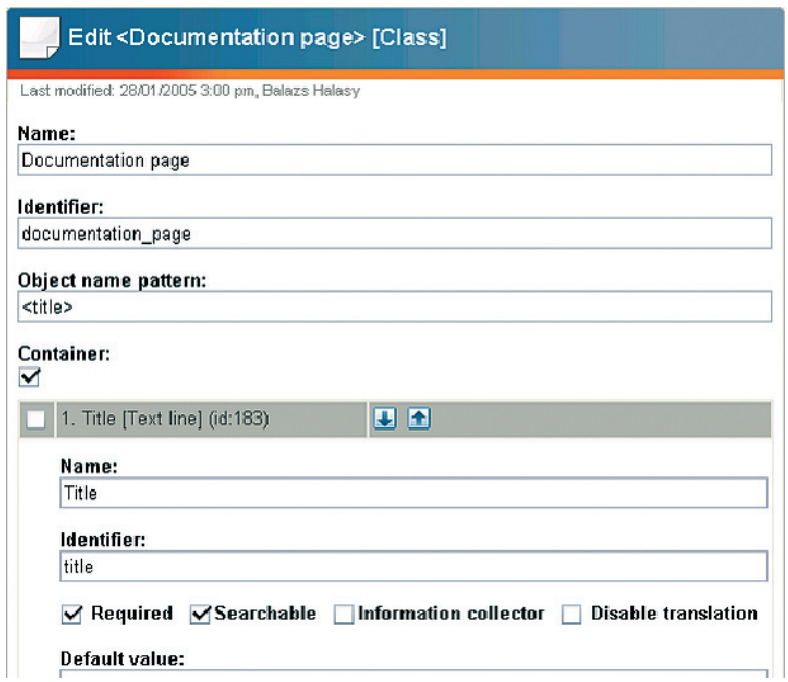


Figure 2.5. The class edit interface

Class structure

A content class consists of the following fields:

- Name
- Identifier
- Object name pattern
- Container flag
- Attributes

Name

The "**Name**" field stores a user-friendly name for the content class. A class name can consist of letters, digits, spaces and special characters. The maximum length is 255 characters. For example, if a class defines a data structure for storing information about graduate students, the

name of the class might be `"Graduate student"`. This name will appear in various lists throughout the Administration Interface, but it will not be used internally by the system. If no name is provided, eZ publish will automatically generate a unique name when the class definition is stored.

Identifier

The **"Identifier**" field is for internal use. However, it should be provided by the site administrator. Class identifiers are used in configuration files, in templates and in PHP code. A class identifier can only consist of lowercase letters, digits and underscores. The maximum length is 50 characters. As an example, the identifier for the class named `"Graduate student"` would probably be `"graduate_student"`. If no identifier is provided, eZ publish will automatically generate a unique identifier when the class definition is stored.

Object name pattern

The "**Object name pattern**" determines how the name of an object (that is, an instance of a class) is generated. The pattern usually consists of attribute identifiers (described later) that tell eZ publish which attributes from the class it should use when generating the name of an object. Each attribute identifier must be placed in angle brackets. Text outside angle brackets is included without interpolation. If no pattern is provided, eZ publish will automatically use the identifier of the first attribute.

Container flag

The "**Container flag**" controls whether an instance of a class is allowed to have sub-items (often called "child nodes" or "children"). This setting only affects the Administration Interface and was added to provide a more convenient environment for administrators and content authors. In other words, it doesn't control any low-level logic, it simply controls the way the graphical user interface behaves.

Attributes

As described earlier, the structure of a content class is defined by the attributes it contains. Content classes contain one or more attributes. Class attributes can be added, removed and rearranged at any time using the Administration Interface. If an attribute is added to a class, it will be added to all instances of that class. If an attribute is removed, it will be removed from all instances. (That is, when an attribute is removed, the associated data is deleted.)

Although it is possible to remove and add attributes using the Administration Interface, in some cases these operations may corrupt the database. This may happen when there are too many instances that need to be updated. If the required processing time exceeds the maximum execution time for the PHP scripts, the process will be interrupted and the database may be left in an inconsistent state. At the time of writing, this problem can only be solved by increasing the maximum execution time of PHP scripts, which is defined in the `max_execution_time` setting in the *`php.ini`* configuration file. The default value is 30 seconds; it should be increased to a couple of minutes. A more reliable solution in the form of a command-line tool for manipulating attributes is planned for future releases of eZ publish.

Class attributes

A content class is made up of one or more attributes, where each attribute is represented by a datatype. The characteristics of an attribute are determined by the datatype chosen for that specific attribute. An attribute is made up of the following fields:

- Name
- Identifier
- Generic controls
- Datatype-specific controls

Name

The **"Name**" field is used for storing a user-friendly name for the attribute. For example, if the attribute is supposed to store birthdates, the name of the attribute would most likely be "Date of birth". This string will appear in various places within the Administration Interface, but it will not be used internally by the system. The name of an attribute can consist of letters, digits, spaces and special characters. The maximum length is 255 characters. If no name is provided, eZ publish will automatically generate a unique name for the attribute when the class definition is stored.

Identifier

The **"Identifier**" of an attribute is for internal use. However, it must be provided by the site administrator. In particular, attribute identifiers are used in configuration files, in templates and in PHP code. An attribute identifier can only consist of lowercase letters, digits and underscores. The maximum length is 50 characters. For example, if the attribute is supposed to store birthdates, the identifier of the attribute would probably be "date_of_birth". If no identifier is provided, eZ publish will automatically generate a unique identifier when the class definition is stored.

Generic controls

Each attribute has a set of generic controls. These controls are the same for each attribute, regardless (but not independent) of the datatype that represents the attribute. Controls can be enabled or disabled. The controls are:

- Required
- Searchable
- Information collector
- Translatable

Required

The **"Required**" switch controls the behavior of the storage procedure for content objects (instances of a content class). It can be used regardless of the datatype that represents the attribute. If the required switch is enabled, the relevant attribute must be set before the content object can be saved. When the required flag of an attribute is set, the system will keep rejecting the input data until all required information is provided. If the required flag is not set, eZ publish will not

care whether or not any data was provided for that attribute. When an attribute is added, the required switch is off. Note that input data will be validated according to the chosen datatype's validation rules, regardless of the state of the attribute's required switch. In other words, even if an attribute is not required, only valid data can be stored in the attribute. Input validation is supported by most (but not all) of the builtin datatypes. The following example demonstrates how these features actually work.

In this scenario, a content class is created that defines a data structure for storing information about prisoners. The class would typically consist of various attributes for storing different kinds of data: name, identification number, date of birth, cell, block, etc. Having at least the name and the birthdate attributes set as required fields will eliminate the possibility of storing convict records without names and / or birthdates. If the birthdate attribute uses the built-in "date" datatype, the system will only accept the input if the birthdate is in a valid date format.

Searchable

The **"Searchable**" switch is used to control whether the data stored for the attribute should be indexed by the search engine or if it should be left unindexed. Refer to the *Datatypes* section of the appendix to see which datatypes support search indexing.

Information collector

Attributes marked as "**Information collectors**" allow users to input data while viewing a page. The information collector switch is used to control the attribute's behavior in View mode. The default View mode behavior is the read-only display of the information that was entered while in Edit mode. For example, when viewing a news article, the contents of the article are displayed but cannot be edited. However, if an attribute is marked as a collector, it will allow information to be entered in View mode. This feature provides site interactivity, for example by being used to quickly create feedback forms. The contents of a form created using this technique will be emailed to the site administrator (or to a specified address) once the form is submitted. Information collection is only supported by a small set of the built-in datatypes (as described in the *Datatypes* section of the appendix). The following example demonstrates how this feature could be used to create a basic feedback form.

Imagine that a content class called `"Feedback form"` is created using the following attributes: name, subject and message. The subject and the message attributes would be marked as information collectors. When an instance of this class is viewed, the subject and the message attributes will be displayed as input fields along with a **Send** button.

Translatable

The **"Translatable**" switch controls whether data stored in the attribute should exist in only one language (the default language) or whether it should be possible to translate the data into additional languages. The translation mechanism is completely independent of the datatype layer. In other words, this switch can be used regardless of the datatype chosen to represent the attribute.

When an attribute is added, the translation switch is switched on by default. Turning it off is typically useful when the attribute is supposed to store nontranslatable input, for example numerical values, prices, e-mail addresses, etc.

Datatype-specific controls

An attribute can have additional controls that are specific to the attribute's datatype. Some datatypes allow fine-grained customization, some do not. For example, the built-in "Text line" datatype provides two settings: default value and maximum length.

The content object

A content object is an instance of a content class. While the content class defines the type and structure of the data, it is the content object that actually stores the data. Once a content class is defined, many instances of that class can be created. For example, if a class for storing news articles is created, several "article" objects (each containing a different news story) can then be instantiated. The following illustration summarizes and shows the relation between datatypes, attributes, content classes and content objects.

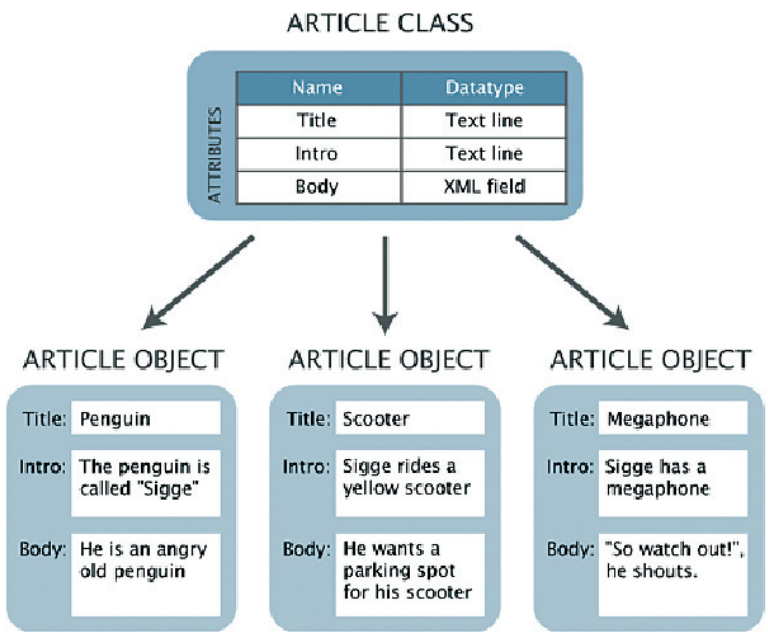


Figure 2.6. Datatypes, attributes, a content class and objects

The illustration is a simplified version of the content model. It doesn't show the exact structure of the objects, as the versioning and the translation layers have been left out. The following section is a more detailed explanation of the object structure. The versioning and the translation layers will be explained in later sections.

Object structure

The following list shows the most important elements of a content object:

- Object ID
- Name

- Type
- Owner
- Creation time
- Modification time
- Status
- Section ID
- Versions
- Current version

Object ID

Every object has a unique identification number. *Object* ID numbers are used by the system to keep track of different objects. ID numbers are not recycled. In other words, if an object is deleted, the ID number of that object will not be reused when a new object is created.

Name

The *Name* of an object is a friendly name that appears in various lists throughout the Administration Interface. It helps the user identify different objects by their names instead of having to deal with identification numbers. An object's name is generated automatically by the system when the object is published. The *object name pattern* definition of a class dictates how objects of that class should be named. This mechanism makes it possible to automatically generate names based on the object's attributes. Since the object name is not used by the system, different objects can have the same name.

The name of the object will be updated every time the object is published. In other words, if attributes specified in the *object name pattern* are changed, the object's name will automatically also be changed. For example, when dealing with news articles, the title of the article would most likely be used to generate the object names. When an article object is published, its name will be a copy of the object's title attribute. When the title is changed, the name of the object will also be updated.

Type

The *Type* information indicates the content class of the object.

Owner

The *Owner* property identifies the user who created the object. This property is set by the system the first time the object is published and never changes (even if the user is removed from the system).

Creation time

The *Published* field contains a timestamp that shows the exact date and time the object was first published. This information is set by the system and cannot be modified.

Modification time

The *Modified* field contains a timestamp that shows the exact date and time the object was last modified. This information may only be set by the system and is updated every time the object is published.

Status

The *Status* f eld indicates the current state of the object. There are three possibilities:

- Draft
- Published
- Archived

When initially created, the object's status is set to *Draft*. This status will remain until the object is published. On publishing, the object's status is set to *Published*. Once published, the object cannot revert to draft status. When a published object is moved to the trash, the status will be set to *Archived*. If a published object is removed from the trash (or removed without being put in the trash first), it will be permanently deleted.

Section

The *Section* ID of an object denotes the section where the object belongs. Each object can belong to one section. By assigning different sections to objects, it is possible to have different groups of objects. The section mechanism is explained later in this chapter.

Versions

The content of an object is stored in multiple *Versions*. A "Version" can be thought of as a timestamped collection of data (the object's attributes) that belongs to a specific user. Every time the contents of an object are edited, the modifications are stored in a new version of the object. All previous versions of the object will remain untouched. This makes it possible to easily revert unwanted or accidental changes. An object always has at least one version. Each version is identified by a number which is automatically incremented for each new version created. The structure and logic of the versioning mechanism is explained in the next section.

Current version

The *Current version* is a number that indicates the current published version of the object. As described above, the contents of an object may exist in several versions. However, only one of them can be the current version (also referred to as the published version). The current version of the object is the version that will be displayed when the object is viewed.

Object versioning

eZ publish comes with a built-in versioning system that is implemented at the object level. This mechanism makes it possible to have several versions of the contents (attributes) of an object. It provides a generic, out-of-the-box version control framework that can be used with any kind of content. Different versions are encapsulated by the object itself. The following illustration shows a more detailed example of the object structure.

Every time an object is edited, a new version of the object's contents will be created. It is always the new version that will be modified - the prior versions are unchanged. This is how eZ publish keeps track of changes made by various users. An accidental or unwanted change can be undone by simply reverting an object to its previous version.

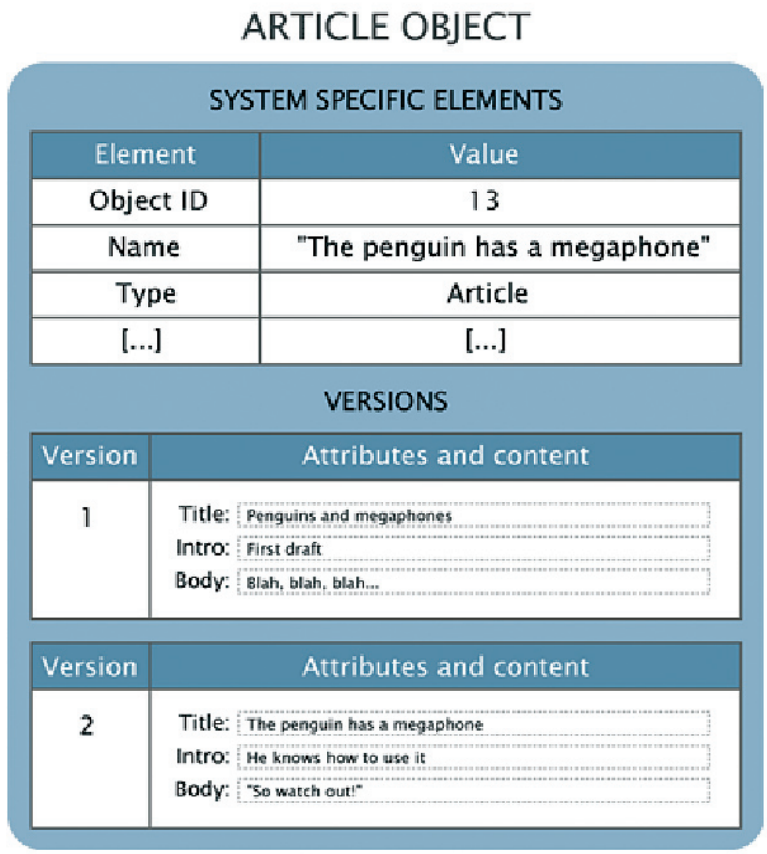


Figure 2.7. Example of a content object that consists of two versions

Version limitations

Since every edit procedure results in the creation of a new version (unlessthe new version is discarded), the database can be quickly filled up by different versions of the same content. In order to prevent this problem, the versioning system can be configured to store a limited number of versions per object. It is possible to assign different version limitations for different object types (that is, different content classes). The default limitation is 10, which means that every object can have a maximum number of 10 versions of its content. If the maximum count is reached, the oldest version will be automatically deleted and a free slot will therefore be available for a new version. This is the default behavior. An alternate setting can be used to disallow the creation of new versions until an existing version is manually deleted by a user.

Version structure

The following list shows the most important elements of a version:

- Version number
- Creation time

- Modification time
- Creator
- Status
- Translations

Version number

Every version has a unique *Version number*. This number is used by the system to organize and keep track of the different versions of an object. The version number is automatically incremented for each version created inside an object.

Creation time

The *Creation time* contains a timestamp indicating the date and time when the version was created. This information is set by the system and will remain the same regardless of what happens to the version.

Modification time

The *Modification time* contains a timestamp of the exact date and time that the version was last modified. This information is updated by the system every time the version is stored and when the version is finally published. When a version is published, the modification time of the object itself will be updated. (It will be set to the same value as the modification time of the version that was published.)

Creator

The version's *Creator* contains a reference to the user who created the version. Although a content object can only belong to a single user (stored in the "Owner" field), each version may belong to a different user. The creator reference is set by the system when the version is created. It cannot be manipulated and will not change, even if the user who created the version is removed from the system.

Status

The state of a version is indicated by its *Status*. There are five possibilities:

- Draft
- Published
- Pending
- Archived
- Rejected

A newly created version is a *Draft*. This status will remain until the version becomes *Published*. Although an object can have many versions, there can only be one published version (the others are usually drafts and archived versions). The published version can be considered the current version and it is the one that is accessed when the object is viewed. A published version cannot become a draft. However, it will become *Archived* as soon as another version is published. The following illustration shows how the versioning system works.

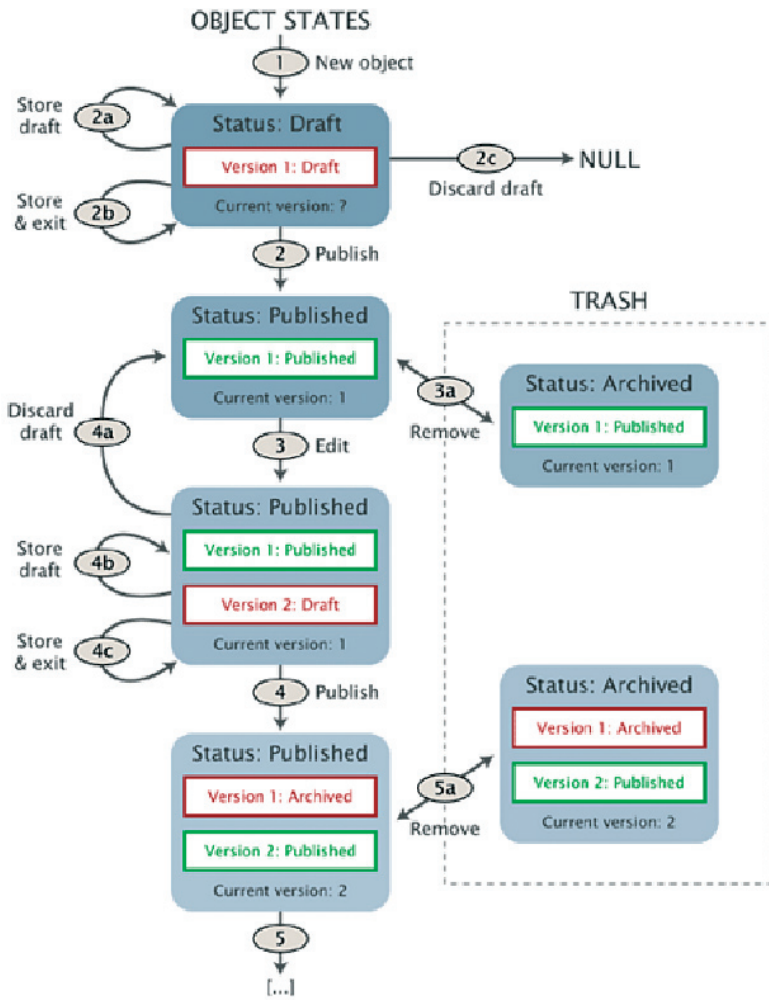


Figure 2.8. Object state overview

The illustration shows the most common states of a content object. When a new object is created (step 1), eZ publish also creates a new draft version. Because the object has not yet been published, its status is set to ”draft” and the current version is unknown. Storing the draft (steps 2a and 2b) will not change the state of the object. The only thing that will happen is that the contents of the draft will be stored in version 1.

If the draft (which is the only existing version) is discarded, the object is completely removed from the system (step 2c). When the draft is published (step 2), the state of both the draft and the object will be set to ”published”. In addition, the current version number will be set to 1, which is the current published version of the object. After publication, the contents of the object can be viewed by site visitors. A published object can be removed or deleted from the system (step 3a). When removed, the object’s state will be set to ”archived” and it will be stored in

the trash. The object can be recovered from the trash to its previous state. Among other things, this involves the status field being set to ”published” again.

When a published object is edited (step 4), the current version (version 1 in this case) will remain unchanged and a completely new version will be created. The contents of the new version (version 2 in this case) will be a copy of the contents of the current version. Again, storing the draft (steps 4b and 4c) will not change the state of the object. If the draft is discarded (step 4a), it will be completely removed from the system and the object will be in the same state it was in before it was edited. If the newly created and edited draft is published, it will become the current version of the object and the previous version (version 1 in this case) will therefore be set to ”archived”. Step 5a illustrates what would happen if the object (now with two versions) was removed.

The remaining states (”pending” and ”rejected”) are used by the eZ publish collaboration functions. When a version is waiting to be approved by an editor, the status is set to ”pending”. If the version is approved, it will be automatically published and thus the status will be set to ”published”. If a pending version is rejected by the editor, the status will be set to ”rejected”.

A version can only be edited if it has a ”draft” status, and it can only be edited by the same user who initially created it. In addition, rejected versions can also be edited. When a rejected version is edited, it will become a draft. Published and archived versions cannot be edited. However, it is possible to make copies of them. When a published or archived version is copied, the status of the copy is set to ”draft” and thus it becomes editable. When the new draft is published, the system automatically sets the status of the previously published version to ”archived” and the new draft will become the published version.

Translations

The contents of each version is stored inside different *translations*. A translation is a representation of the information in a specific language. In other words, the translation layer allows a version of the object’s contents to exist in different languages. A version always has at least one translation of the content.

Multiple languages

In addition to the versioning system, the content model of eZ publish provides a built-in multilanguage framework. This feature was enhanced in eZ publish 3.8. In previous versions, the system kept all translations inside one version and would allow only one person to do modifications. This made it hard for several persons to edit the translations separately at the same time.

In 3.8, translations have become more self-contained and independent. They can be created and edited separately by multiple users, meaning that multiple translators can work with the same content. The following list shows the most important multilanguage changes in eZ publish 3.8:

- Users can only edit one version and translation at a time.
- Objects can be created in any language for any siteaccess. The objects do not have to exist in the primary language before they can be translated using the additional languages.
- Objects can be translated to as many languages as necessary. Several translators can work on the same object at the same time.
- Available translations can be controlled per siteaccess. Translations can be filtered.

Implementation

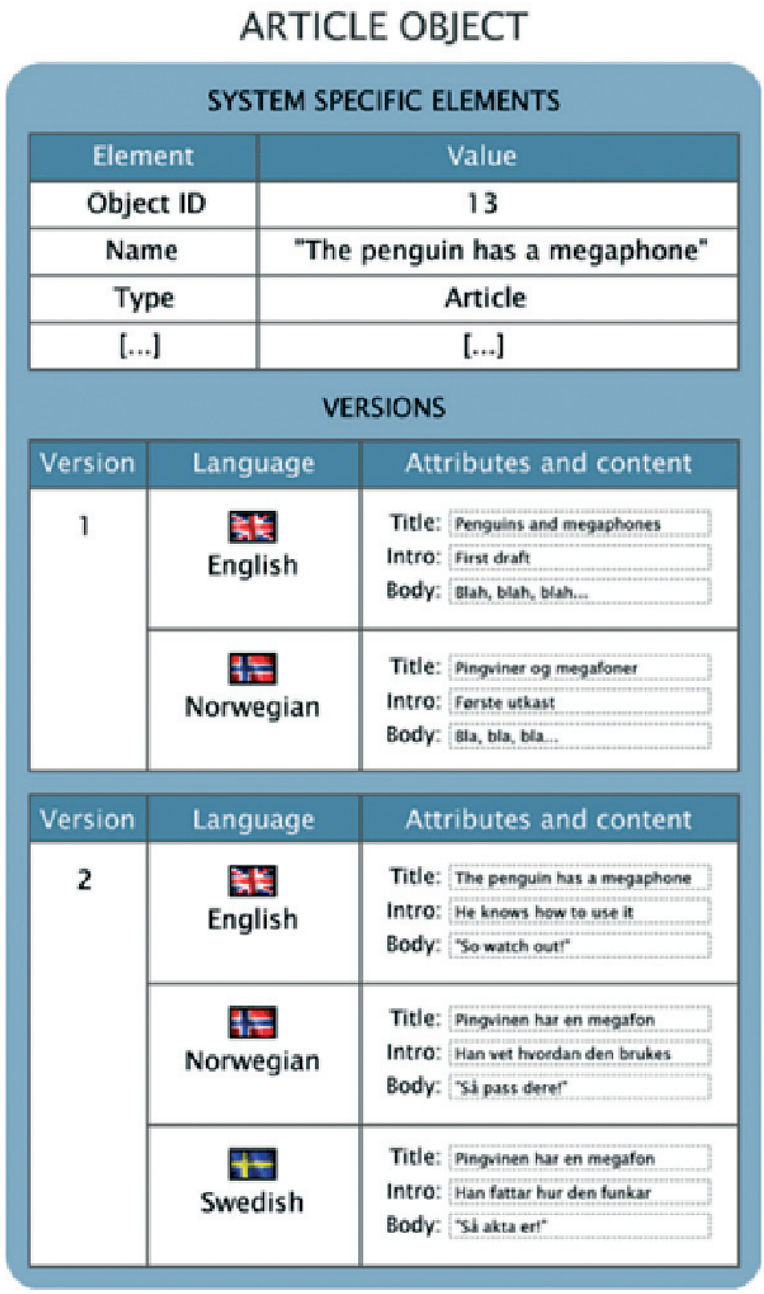


Figure 2.9. Content object structure (with versions and translations)

The multilanguage layer is implemented at the version level and thus it allows a version to exist in several languages. It provides a generic one-to-one translation mechanism that can be used to translate any kind of content. A one-to-one translation solution makes it possible to represent the same content using multiple languages. For example, when the same news article is available in English, Norwegian and Hungarian, we say that we have oneto-one translation of the content.

The translation mechanism is completely independent of the datatypes. In other words, any kind of content can be translated regardless of the datatypes used to express the content's structure. Therefore, it is possible to start with one language and later add translations, thus extending the spectrum of the target audience. The following illustration shows an example of an object seen by users. The object has three versions and each version exists in several languages. A language in this case is referred to as a *translation*.

As the illustration indicates, each version can have a different set of translations. At a minimum, a version always has one translation, which is the default translation. The default translation of a version cannot be removed. However, additional translations can be added and removed while the version is being edited. A translation for a specific language can only be added if it exists in the *global content translation list* (found in the Setup section of the Administration Interface). This list keeps track of the languages that users are allowed to use when translating content. For example, if the global translation list English, Norwegian and Hungarian, the system will allow those three languages to be used when working with content. Note that in 3.8 an object may be created using any language and thus the "default translation" and the "primary language" terms have become obsolete. It is now possible to remove the translation which was used when the object was created (because the main language of an object can be changed at any time using the Administration Interface).

The global translation list can be changed at any time. A translation added to the global translation list will immediately become available for use. Removing a language from the global translation list will (upon confirmation) also result in the removal of all translations that use that language. Note that in version 3.8, the system will not allow you to remove a language if it is the main (initial) language for some of the objects.

Non-translatable attributes

The data structure defined by a class is built up of attributes, where each attribute is represented by a datatype. Among other things, an attribute of a class can be made translatable or not. If an attribute is translatable, the system will allow the translation of the contents of the attribute when an object of that class is edited. This is typically convenient when the attribute contains text. For example, the written part of a news article can be translated into different languages. However, some attributes are non-translatable by definition. This is typical for numbers, dates, e-mail addresses, images without text, and so on. Such attributes can be made non-translatable and their contents will therefore be copied from the default translation. The copied values cannot be edited.

For example, say a user needs to store information about furniture in multiple languages. He could build a furniture class using the following attributes: name, photo, description, width, height, depth and weight. Allowing the translation of anything other than the description attribute

would be pointless since the values stored by the other attributes are the same regardless of the language used to describe the furniture. In other words, the name, photo, width, height, depth and weight would be the same in, for example, both English and Norwegian. Conversion between different measuring units could be done within the template used to display the information.

Access control

It is possible to control whether or not a user (or a group of users) is able to translate content. This policy can be controlled on a class, section and owner basis. In 3.8 there is now also a fine-grained mechanism for controlling access to the different languages / translations. In addition, it is possible to control access to the global translation list. This allows users other than the site administrator to add and remove translations on a global basis.

The content node

When the system is in use, new content objects are created on the fly. For example, when a new article is composed, a new article object is created. Obviously, the content objects can't just hover around in space, they have to be organized in some way. This is where the *nodes* and the *content node tree* comes in. A content node is nothing more than an encapsulation of a content object. In eZ publish, every object is usually represented by one or more nodes. The following illustration shows a simplified example of a node and a corresponding object (which is referenced by the node) as it would have been represented inside the system.

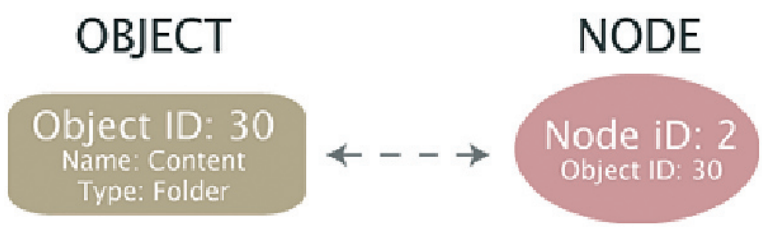


Figure 2.10. Object - node relation

The content node tree is built up of nodes. A node is simply the location of an object within the tree structure. The tree is the actual mechanism used to hierarchically organize the objects that are present in the system. The content node tree is explained in the next section.

Node structure

The following list shows the most important elements of a content node:

- Node ID
- Parent node ID
- Object ID
- Sort method
- Sort order
- Priority

Node ID

Every node has a unique identification number. The *Node ID* numbers are used by the system to organize and keep track of the nodes. These ID numbers are not recycled. In other words, if a node is deleted, the ID number of that node will not be reused when a new node is created.

Parent node ID

The *Parent node ID* of a node reveals the node's superior node in the tree.

Object ID

Every object that exists in the system has a unique identification number. The *Object ID* of a node pinpoints the actual object which the node encapsulates.

Sort method

The *Sort method* of a node determines how the children of the node should be sorted. The following sorting methods are available:

Method	ID	Description
Class identifier	6	The nodes are sorted by the class identifiers of the objects.
Class name	7	The nodes are sorted by the class names of the objects.
Depth	5	The nodes are sorted by their depth within the tree. A node further down in the tree has a higher level of depth. The root node has a depth of 1.
Modified	3	The nodes are sorted by the modification time of the objects.
Modified subnode	10	The nodes are sorted by the modification time of their children.
Name	9	The nodes are sorted by the names of the objects.
Path	1	The nodes are sorted by their path strings.
Priority	8	The nodes are sorted by their priority. Every node has a priority field that can be set by the user. This solution allows the nodes to be sorted in a custom order. The priority field is described below.
Published	2	The nodes are sorted by the creation time of the object's current / published versions.
Section	4	The nodes are sorted by the section IDs of the objects.

Note that it is possible to combine the sorting methods in order to sort nodes in a more complex way. However, since a node is incapable of ”remembering”a combination and you can only set one method and one order for each node, complex sorting must be done via templates.

Sort order

The *Sort order* determines the order in which the children of the node should be sorted. There are two possibilities:

- Descending (0 / FALSE)
- Ascending (1 / TRUE)

For example, if the sorting method is set to ”Name” and the sort method is set to ”Ascending”, the underlying nodes will be alphabetically sorted from A to Z. If the sort method is set to ”Descending”, the underlying nodes will be sorted from Z to A.

Priority

The *Priority* field allows a user to assign both positive and negative integer values to a node (zero is also allowed). This field makes it possible to sort nodes in a custom way. If the sorting method of a node is set to Priority, the children of that node will be sorted by their assigned integer values.

The content node tree

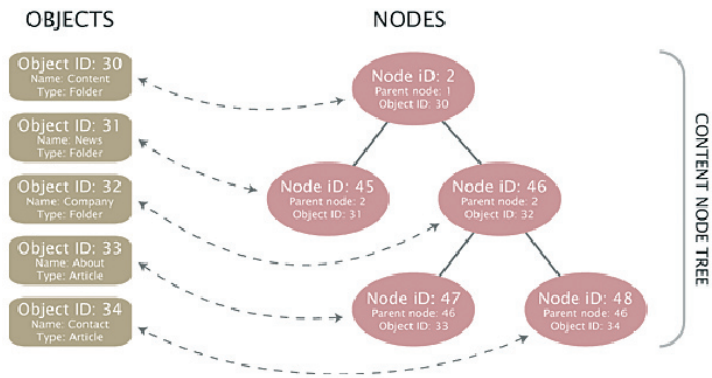


Figure 2.11. Objects, nodes and the content node tree

The following illustration shows the same node structure seen from the user's perspective.

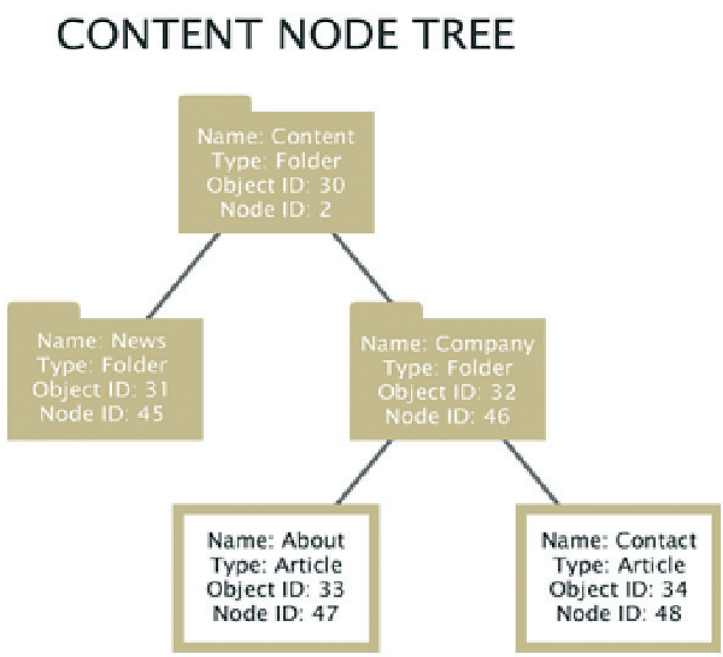


Figure 2.12. Content node tree

The content node tree is a hierarchical organization of objects. Each leaf in the tree is a node (also known as a location). Each node refers to one object. The usual case is that an object is referenced by only one node. Because of the node-encapsulation of objects, any type of content object can be placed anywhere in the tree. At a minimum, the tree consists of one node, called the *root* node. The identification number of the root node is 1. The root node is a virtual node, it does not encapsulate an actual object and it cannot be deleted. A node that is directly below the root node is called a *top-level* node (the top-level nodes are described in the next section). The depth and width of the tree is virtually unlimited. The following illustration shows a simplified example of how objects are referenced by nodes which together make up a content node tree.

Multiple locations

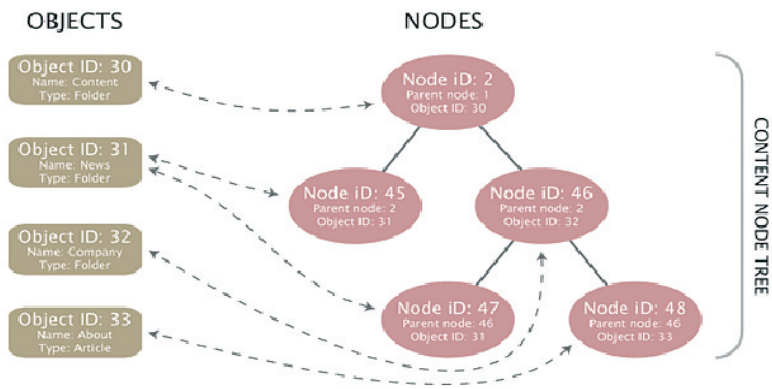


Figure 2.13. Objects, nodes and the content node tree - multiple locations

The following illustration shows the same node structure seen from the user's perspective.

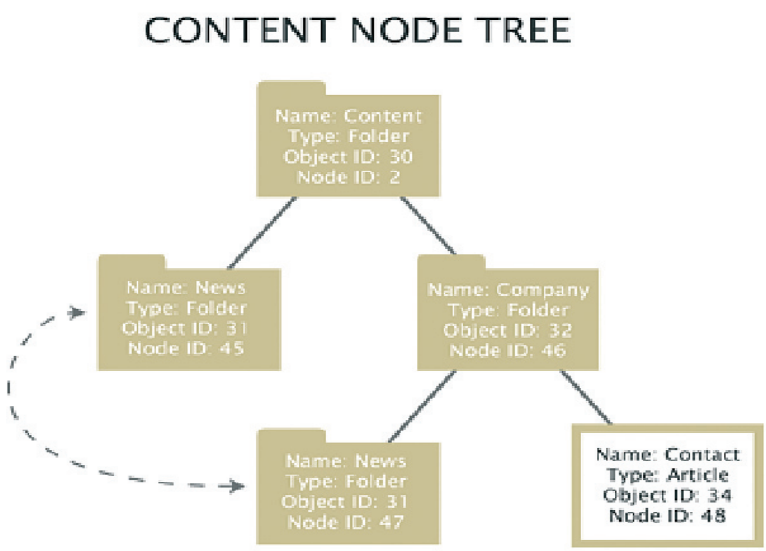


Figure 2.14. Content node tree with multiple locations

An object may be referenced by several nodes, which means that the same object can appear at different locations within the tree. This feature can for example be used to place a specific news article at two locations: the front page and the archive. When an object has multiple nodes / locations, only one node can be considered the main node of the object. The main node usually represents the object's original location in the tree. The other nodes can be thought of as additional nodes / locations. Among other things, the main node is used to avoid duplicate search hits, infinite recursive loops, smart filtering, etc. The following illustration shows an example of a structure where an object has multiple locations in the tree. The locations may have different sets of sub items.

Warning

A very common mistake when planning the structure of a site is thinking of multiple locations as shortcuts or links on a filesystem. Unfortunately, this is not how the node tree works. When a new location is added to an object, eZ publish will not go through and replicate the node structure below the object's original location. For example, if a folder containing several subfolders with articles, images, etc. is assigned a secondary location, the subfolders with articles, images, etc. will not be available below the new location of the folder.

Additional notes

Only published objects appear in a content node tree. A newly created object (with its status set to ”draft”) will not be assigned a node until the object is published for the first time. An object is considered to be deleted (status set to ”archived”) when all nodes referencing that object are removed from the tree. A deleted object will appear in the system trash. It is important to understand that the trash in eZ publish is a flat structure. This is different from what people are used to from the trash implementation in modern operating systems. When an object is to be recovered from the trash, it must be manually placed in the tree since the deleted object doesn't contain any information about its previous location. For example, if a folder containing some news articles is deleted, both the folder and the articles it holds will appear on the same level within the trash. Recovering the folder itself will not bring back the articles since the links between the folder and the articles were lost when the nodes were deleted. (Note that this behavior was changed in eZ publish 3.8. It is now possible to recover the deleted objects' locations.)

Top-level nodes

A typical eZ publish installation comes with the following set of top-level nodes:

- Content
- Media
- Users
- Setup

The top-level nodes cannot be deleted. However, they can be swapped with other nodes. The swap function can be used to change the type of a top-level node. For example, the ”Content” node references a folder object. By swapping it with another node which references a different kind of object, it is possible to change the type of the top-level node itself. The following illustration shows the virtual root node and the standard top-level nodes:

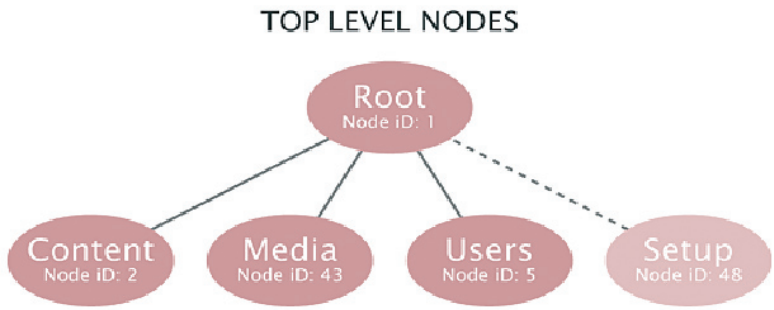


Figure 2.15. Top-level nodes

Content

The content of a site is placed under the *Content* node. This node is typically used for organizing folders, articles, information pages, etc. and thus defines the content structure of the site. A sitemap can be easily created by traversing the contents of this top-level node. The default identification number of the Content node is 2. The contents of this node can be viewed by selecting the ”Content structure” tab in the Administration Interface. By default, this node references a *Folder* object.

Media

The *Media* node is typically used for storing and organizing information that is frequently used by the nodes located below the Content node. It usually contains images, animations, documents and other files. For example, it can be used to create an image repository containing images used in different news articles. The default identification number of the Media node is 43. The contents of this node can be viewed by selecting the ”Media library” tab in the Administration Interface. By default, this node references a Folder object.

Users

The built-in multi-user solution makes use of the native content structure of eZ publish. Users are simply instances of classes containing the ”User account” datatype. The user nodes are organized within *User group* nodes below the *Users* top-level node. In other words, this node contains the actual user accounts and groups. The default identification number of the Users node is 5. The contents of this node can be viewed by selecting the ”User accounts” tab in the Administration Interface. By default, this node references a User group object.

Setup

The *Setup* node contains miscellaneous nodes related to configuration and is used internally. The default identification number of the Setup node is 48. By default, this node references a Folder object.

Node visibility

Since publishing means adding an object (by the way of a node) to the content tree, unpublishing would imply the removal of the object from the tree. Once an object is published, it cannot be unpublished because eZ publish does not provide such a feature. Instead, the system provides a hiding mechanism which can be used to change the visibility of nodes. The ”hide” feature makes it possible to prevent the system from displaying published objects. This is achieved by denying access to the nodes. A single node or a subtree of nodes can be hidden by either users or the system. The visibility status of a node may be one of the following:

- Visible
- Hidden
- Hidden by superior

All nodes are visible by default and thus the objects they reference can be accessed. User can hide or reveal nodes using the Administration Interface. Once a node is hidden, all its descendants will automatically be marked ”Hidden by superior,” and therefore the descendants will also become hidden. A node cannot become visible if its parent is hidden.

A hidden node will not be available unless the `ShowHiddenNodes` directive within the `[SiteAccessSettings]` block of a configuration override for *`site.ini`* is set to true. The most common way to use this setting is to allow only the Administration Interface to show hidden nodes. This means that the hidden nodes will not be visible on the actual site.

Implementation

Each node has two flags: ”H” and ”X”. While ”H” means ”hidden”, ”X” means ”invisible”. The hidden flag reveals whether the node has been hidden by a user. A raised invisibility flag means that the node is invisible either because it was hidden by a user or by the system. Together, the flags represent the three visibility states that were described above:

H	X	Status
-	-	The node is visible.
1	1	The node is invisible. It was hidden by a user.
-	1	The node is invisible. It was hidden by the system because its ancestor is hidden / invisible.

If a user tries to hide an already-invisible node, the node’s hidden flag will be set in addition to the invisible flag. If a node is hidden and its parent becomes visible, the node will remain hidden while its descendants remain invisible. The following illustrations show how the node hiding algorithm works.

Case 1: Hiding a visible node

The next illustration shows what happens when a visible node is hidden by a user. The node will be marked hidden. Underlying nodes will be marked invisible (hidden by superior). The visibility status of underlying nodes already marked hidden or invisible will not be changed.

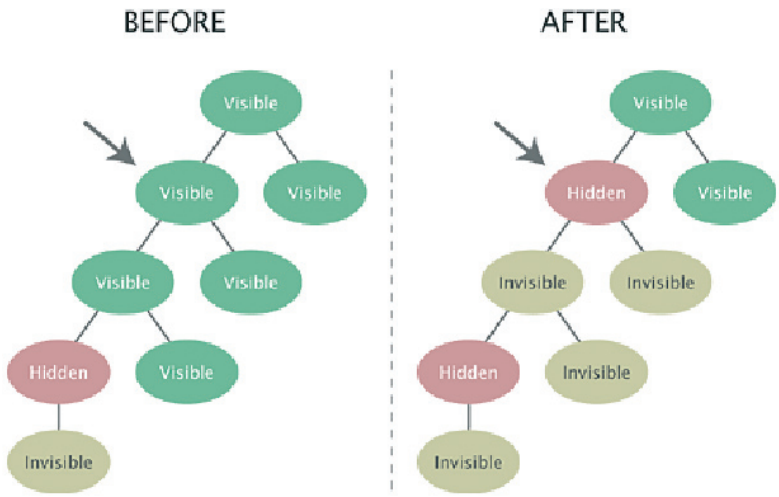


Figure 2.16. Hiding a visible node

Case 2: Hiding an invisible node

The following illustration shows what happens when an invisible node (hidden by superior) is explicitly hidden by a user. The node will be marked as hidden. Since the underlying nodes are already either hidden or invisible, their visibility status will not be changed.

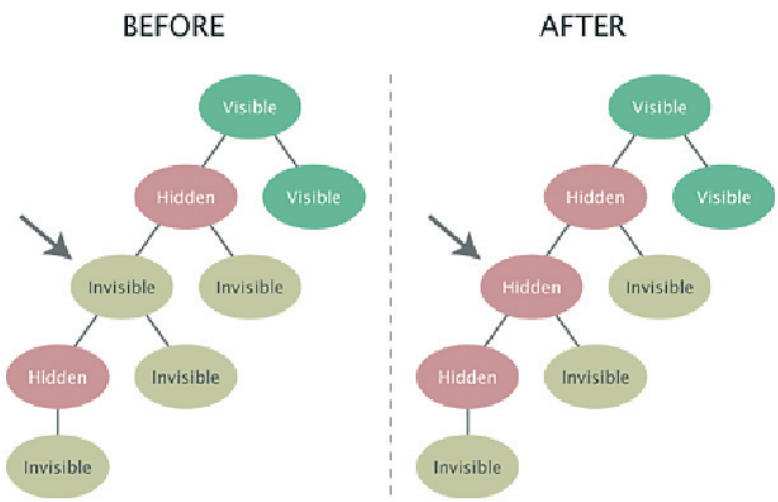


Figure 2.17. Hiding an invisible node

Case 3: Revealing a node with a visible ancestor

The next illustration shows what happens when a user reveals a node that has a visible ancestor. Underlying invisible nodes will become visible. An underlying node that was explicitly hidden by a user will remain hidden (and its children will remain invisible).

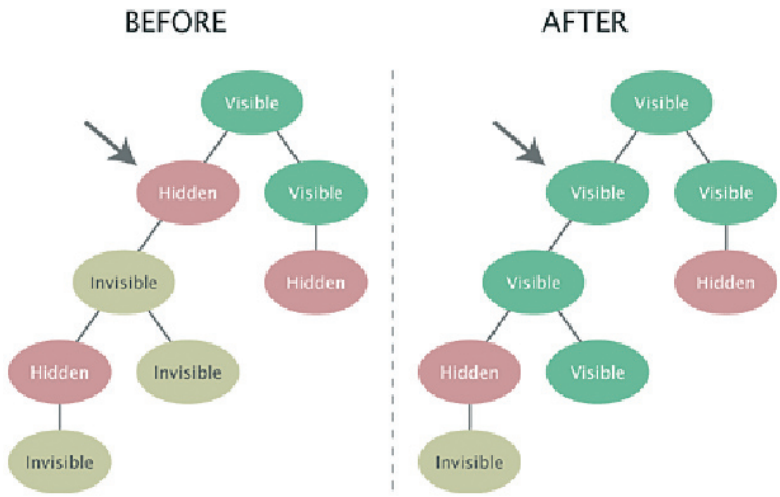


Figure 2.18. Revealing a node with an invisible ancestor

Case 4: Revealing a node with an invisible ancestor

The following illustration shows what happens when a user reveals a node that has an invisible ancestor. Since the target node is revealed in a subtree that is currently invisible (because a node further up in the hierarchy has been explicitly hidden), the node will not become visible. Instead, it will be marked as invisible and will become visible when the hidden superior node is revealed.

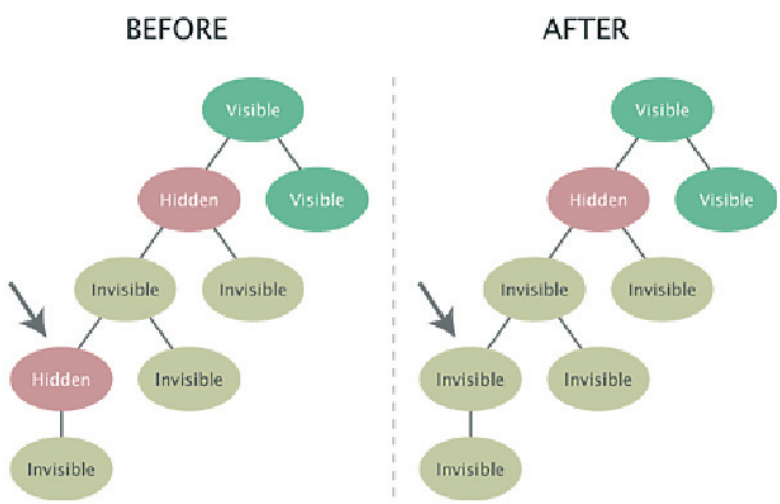


Figure 2.19. Revealing a node with an invisible ancestor

Sections

A *section ID* is a number that can be assigned to an object. The section ID of an object denotes which section the object belongs to. Each object can belong to one section. By assigning different sections to objects, it is possible to have different groups of objects. Although the sectioning mechanism is implemented at the object level, it is likely to be used in conjunction with the content node tree. This is why the Administration Interface makes it possible to manage sections at the node level. Using sections makes it possible to:

- Segment the node tree into different subtrees
- Set up custom template override rules
- Limit and control access to content
- Assign discount rules to a group of products

A default eZ publish installation comes with the following sections:

ID	Name	Description
1	Standard	The *Standard* section is the default section. The top-level Content node makes use of this section.
2	Users	The *Users* section is dedicated for user accounts and user groups. The top-level Users node makes use of this section.
3	Media	The *Media* section is used by the top-level Media node.
4	Setup	The *Setup* section is used by the top-level Setup node.

Section definitions can be added, modified and removed using the Administration Interface. The following illustration shows an example of how this feature can be used to segment the content node tree.

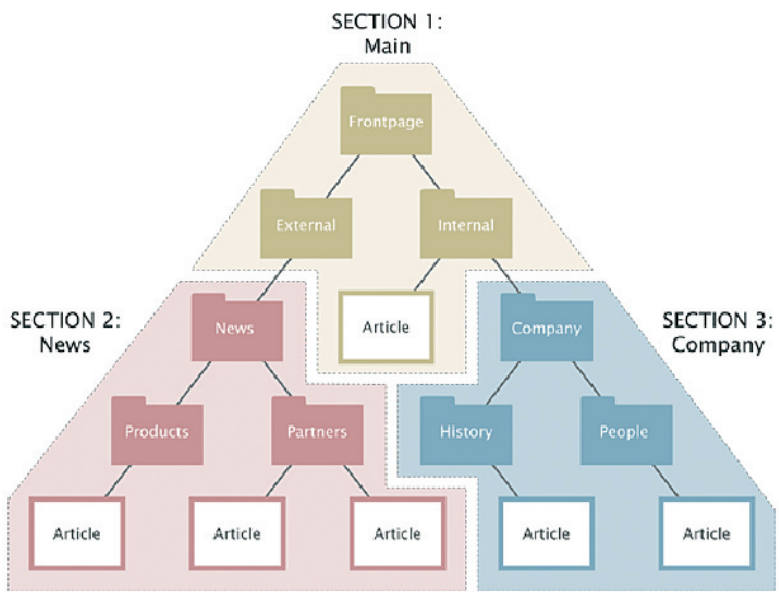


Figure 2.20. Example of sections

Behavior

When a new object is created, its section ID will be set to the default section (which is usually the ”Standard” section). When the object is published, it will automatically inherit the section assigned to the object encapsulated by its parent node. For example, if an object is created in a folder that belongs to section 13, the section ID of the newly created object will be set to 13. If an object has multiple node assignments, then the section ID of the object referenced by the parent of the main node will be used. If the main node of an object with multiple node assignments is changed, the section ID of that object will be updated.

The Administration Interface makes it possible to assign sections to objects using the node tree. When a section is assigned to a node, the section ID of the object referenced by that node will be updated. In addition, the section assignment of all subsequent children of that node will also be changed. For example, if the section ID of a folder containing news articles is changed, then the section ID of the articles in that folder will also be changed.

The removal of sections may corrupt permission settings, template output and other things in the system. In other words, a section should only be removed if it is completely unused. When a section is removed, it is only the section definition itself that will be removed. Other references to the section will remain and thus the system will most likely be in an inconsistent state.

The section ID numbers are not recycled. If a section is removed, the ID number of that section will not be reused when a new section is created.

URL storage

Every address that is input as a link into an attribute using the `XML block` and URL datatypes is stored in a separate part of the database. Data stored using these datatypes only contains references to entries in the separate URL table. This feature makes it possible to inspect and edit the published URLs without having to interact with the content objects. The addresses in the URL table can be checked by running the *`linkcheck.php`* script (which is also executed by the cronjob script) that comes with eZ publish. This script will check whether the links in the table actually work by accessing them one by one. If the target server of a URL returns an invalid response (”404 Page not found”, ”500 Internal Server Error”, ”403 Access Denied”, etc.) or if there is no response, the URL will be marked invalid. Invalid URLs and the objects which use them can be easily filtered out and edited using the ”URL management” section of the Administration Interface. An entry in the URL table consists of the following data:

- ID
- Address
- Creation time
- Modification time
- Last checked
- Status

Every URL has a unique identification number (*ID*). The address contains the actual link. The creation time is the exact date / time when the object containing that URL was published. The modification time is updated every time the URL is changed using the ”URL management” part

of the Administration Interface (and not when the object containing that URL is edited) Whenever a URL is checked by the script, the ”last checked” field will be updated. The status of a URL can be either ”valid” or ”invalid”. By default, all URLs are valid. When the cronjob or the link-check script is running, it will automatically update the status of the URLs. If a broken link is found, its status will be set to ”invalid”. Whenever an already existing URL is stored, the system will simply reuse the existing entry in the table.

Tip

The link check script must be able to contact the outside world through port 80. In other words, the firewall must be opened for outgoing HTTP traffic from the web server that is running eZ publish.

Information collection

The information collection feature makes it possible to gather user input when a node referencing an information collector object is viewed. It is typically useful when it comes to the creation of feedback forms, polls, etc.

An object can collect information if at least one of its class attributes is marked as an information collector. When the object is viewed, each collector attribute will be displayed using the chosen datatype’s data collector template. Instead of just outputting the attributes’ contents, the collector templates provide interfaces for data input. The input interface generated depends on the datatype which represents the attribute. The following table shows the datatypes capable of collecting information. (Note that in version 3.8 some of the other datatypes have been upgraded to support information collection.)

Datatype	Input interface	Input validation
Checkbox	Checkbox.	No.
E-Mail	Single line of text.	Yes.
Option	Radio buttons or a dropdown menu.	No.
Text block	Multiple lines of unformatted text.	No.
Text line	Single line of unformatted text.	No.

(Note that information collection support has been added to even more datatypes in eZ publish 3.8.)

The input interfaces are defined within an HTML form that posts the data to `"/content/action"` (the `"action"` view of the `"content"` module), using a submit button named `"ActionCollectInformation"`. The IDs of the node and the object must be provided as hidden variables. The submitted data will be stored in a dedicated part of the database, separate from but related to the object itself. In addition, whenever the object collects any data, the information can be sent to a specified e-mail address.

Tip

The **Collected information** section within the **Setup** section of the Administration Interface can be used to view and delete information that was collected through content objects.

Configuration

This section explains the configuration model of eZ publish. The default configuration files end with an *.ini* extension and are located in the *settings/* directory. Each file controls the behavior of a specific part of the system. For example, the *content.ini* file controls the behavior of the content engine, the *webdav.ini* file controls the behavior of the Web-DAV feature, and so on. The main configuration file is called *site.ini*. Among other things, it tells eZ publish which database, design, etc. should be used. The default configuration files contain all the possible directives (with default settings) along with brief explanations. These files should only be used for reference; they should never be modified. (The ”Configuration files” section of the online reference documentation contains a comprehensive explanation of the different configuration files and their settings.)

File structure

An eZ publish configuration file is divided into blocks. Each block contains a collection of settings. The following example shows a part of the main (*site.ini*) configuration file.

```
...
# This line contains a comment.
[DatabaseSettings]
Server=localhost
User=allman
Password=DeLorean
Socket=disabled
SQLOutput=enabled

# This line contains another comment.
[ExtensionSettings]
ActiveExtensions[]=ezdhtml
ActiveExtensions[]=ezpaypal
...
```

The example above shows two blocks: `DatabaseSettings` and `ExtensionSettings`. Each block has several settings which control the behavior of the system. A setting can usually be set to enabled or disabled, a string of text, or an array of strings. If the name of a setting ends with a pair of square brackets, it means that the setting accepts an array of values. In the example above, the `ActiveExtensions` setting tells eZ publish to use two different extensions: *ezdhtml* and *paypal*. Lines starting with a hash (`"#"`) are treated as comments.

Configuration overrides

As pointed out earlier, the default configuration files should never be modified because they will most likely be overwritten by a new set of files during an upgrade. Making a backup will not be sufficient because the configuration settings change over time. For example, the current version of the files will not contain settings that are added in the next release. Because of these issues, custom configuration settings must be placed elsewhere. Global configuration overrides can be placed in the *settings/override/* directory. The settings of the configuration files

located in this directory will override all other settings. The name of the configuration files in the override directory must end with one of the following extensions:

- *.ini.append*
- *.ini.append.php*

If override configuration files exist with both *.ini.append* and *.ini.append.php* extensions, eZ publish will process the one that ends with *.php*. Because of possible security issues, the latter (*.ini.append.php*) should be used; especially if eZ publish is running in a non-virtual host environment. The *.php* extension will trick the web server into handling the configuration file as a PHP script. If someone attempts to read it using a browser, the server will not display the contents. Instead, it will attempt to process it as PHP code. The result will be no output. This method makes it more difficult for hackers to get access to the configuration settings (for example the database password) when eZ publish is running in a non-protected (usually non-virtual host) environment. In order for this to work, the contents of the configuration file must be enclosed by a pair of PHP comment markers: /* and */. The following example shows how an override (for example *test.ini.append.php*) should be configured

```
<?php /* #?ini charset="iso-8859-1"?

# These are only example settings
[ExampleSettings]
ExampleSettingOne=enabled
ExampleSettingTwo=disabled
...
*/ ?>
```

The charset directive reveals the character set used to construct the configuration file (usually ISO-8859-1).

Site management

A single eZ publish installation is capable of hosting multiple sites by making use of the *siteaccess* system. This system makes it possible to use different configuration settings based on a set of rules. The rules control the group of settings that should be used in a particular case. The siteaccess rules must be specified in the global override for the *site.ini* configuration file (*settings/override/site.ini.append.php*).

Siteaccess

A collection of configuration settings is called a *siteaccess*. When a siteaccess is in use, the default configuration settings will be overridden by the settings defined for the siteaccess. Among other things, a siteaccess dictates which database, design and var directory should be used. (These components are usually referred to as "resources".) By making use of different siteaccesses, it is possible to combine different content and designs. A typical eZ publish site consists of two siteaccesses: a public interface for visitors and a protected interface for administrators. Both siteaccesses use the same content (same database and same *var/* directory) but use different designs. While the administration siteaccess would most likely use the built-in administration design, the public siteaccess would probably use a custom design. The following illustration shows this scenario.

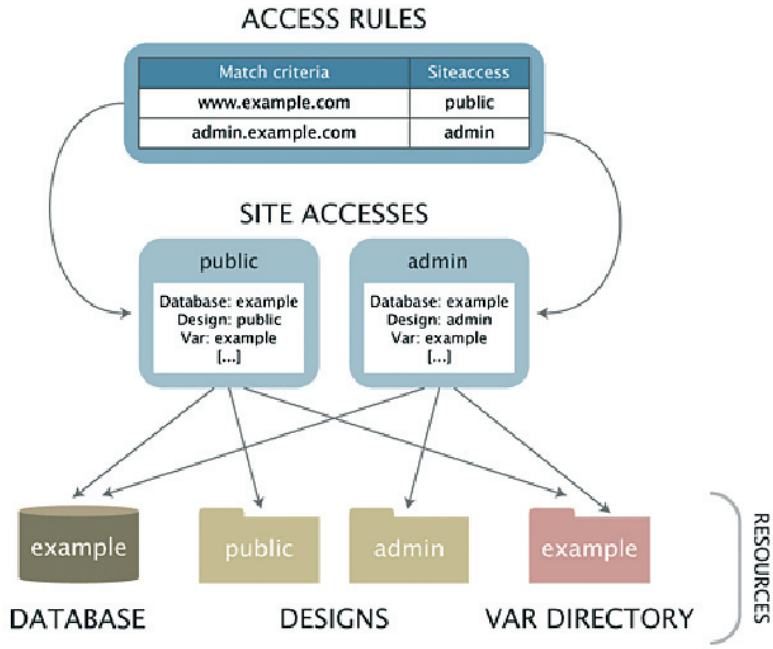


Figure 2.21. Example of a setup with two siteaccesses

Again, a siteaccess is a set of configuration files that override the default settings. A single eZ publish installation can host a virtually unlimited number of sites by using different siteaccesses. The configuration settings for a siteaccess are located inside a dedicated subdirectory within the *settings/ siteaccess/* directory. The name of the subdirectory is the actual name of the siteaccess. The following illustration shows a setup with two siteaccesses: ”admin” and ”public”.

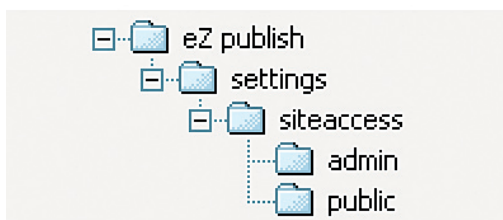


Figure 2.22. Siteaccess directory example

When a siteaccess is in use, eZ publish reads the configuration files using the following sequence:

1. Default configuration settings (*settings/*.ini*)
2. Siteaccess settings (*settings/siteaccess/[name_of_siteaccess]/ *.ini.append.php*)
3. Global overrides (*settings/override/*.ini.append.php*)

In other words, eZ publish will first read the default configuration settings. Next, it will determine which siteaccess to use based on the rules defined in the global override for *site.ini* (*settings/override/site.ini.append.php*). When it knows which siteaccess to use, it will go to the directory for that siteaccess and read the configuration files. The settings of the siteaccess will override the default configuration settings. For example, if the siteaccess uses a database called "Amiga500", the system will see this and automatically use the specified database when an incoming request is processed. Finally, eZ publish reads the configuration files in the global override directory. The settings in the global override directory will override all other settings. In other words, if a database called "CD32" is specified in the global override for *site.ini* then eZ publish will attempt to use that database regardless of what is specified in the siteaccess settings (in this case, the "CD32" database would be used for all incoming requests). If a setting is not overridden by either the siteaccess or from within a global override then the default setting will be used. The default settings are set in the **.ini* files located in the *settings/* directory.

Access methods

Based on a set of rules, eZ publish determines which siteaccess it should use every time it processes an incoming request. The rules must be set up in the global override for the *site.ini* configuration file: *settings/override/site.ini.append.php*. The behavior of the siteaccess system is controlled by the `MatchOrder` setting within the `[SiteAccessSettings]` block. This setting controls the way eZ publish interprets incoming requests. There are three possible methods:

- URI
- Host
- Port

The following text gives a brief explanation of the different access methods. Note that the access methods can be combined by providing the desired methods and separating them using semicolons in the global configuration override for *site.ini*.

URI

This is the default setting for the `MatchOrder` directive. When the *URI* access method is used, the name of the target siteaccess will be the first parameter that comes after the *index.php* part of the requested URL. For example, the following URL will tell eZ publish to use the "admin" siteaccess: `http://www.example.com/index.php/admin`. If another siteaccess with the name "public" exists, then it would be possible to reach it by pointing the browser to the following address: `http://www.example.com/index.php/public`. If the last part of the URL is omitted then the default siteaccess will be used. The default siteaccess is defined by the `DefaultAccess` setting within the `[SiteSettings]` block. The following example shows how to set up *settings/override/site.ini.append.php* in order to make eZ publish use the URI access method and to use a siteaccess called "public" by default:

```
...
[SiteSettings]
DefaultAccess=public
```

```
[SiteAccessSettings]
MatchOrder=uri
...
```

The URI access method is typically useful for testing and demonstration purposes. It doesn't require any configuration of the web server and the DNS server.

Host

The *host* access method makes it possible to map different host / domain combinations to different siteaccesses. This access method requires configuration outside eZ publish. First of all, the DNS server must be configured to resolve the desired host / domain combinations to the IP address of the web server. Secondly, the web server must be configured to trigger a virtual host configuration (unless eZ publish is located in the main document root). Once the DNS server and the web server are configured properly, eZ publish can be set up to use different siteaccesses based on the host / domain combinations of the incoming requests. The following example shows how to set up *settings/override/site.ini.append.php* in order to make eZ publish use the host access method. It also shows the basic usage of the host matching mechanism.

```
...
[SiteAccessSettings]
MatchOrder=host
HostMatchType=map
HostMatchMapItems[]=www.example.com;public
HostMatchMapItems[]=admin.example.com;admin
...
```

The example above tells eZ publish to use the "public" siteaccess if the requested URL starts with `"www.example.com"`. In other words, the configuration files in *settings/siteaccess/public/* will be used. If the requested URL starts with `"admin.example.com"`, then the admin siteaccess will be used. The example above demonstrates only a fragment of the host matching capabilities of eZ publish. Refer to the online documentation for a full explanation of the `HostMatchType` directive.

Port

The port access method makes it possible to map different ports to different siteaccesses. This access method requires configuration outside eZ publish. The web server must be configured to listen to the desired ports (by default, a web server typically listens for requests on port 80, the standard port for HTTP traffic). In addition, the firewall will probably have to be opened so that the incoming traffic actually reaches the web server. The following example shows how to configure *settings/override/site.ini.append.php* in order to make eZ publish use the port access method. It also shows how to map different ports to different siteaccesses.

```
...
[SiteAccessSettings]
MatchOrder=port
```

```
[PortAccessSettings]
80=public
81=admin
...
```

The example above tells eZ publish to use the ”public” siteaccess if the requested URL is sent to the web server using port 80. In other words, the configuration files inside *settings/siteaccess/public/* will be used. If the URL is requested on port 81 (usually by appending a :81 to the URL, for example `"http://www.example.com:81"`), the admin siteaccess will be used.

Modules and views

The *modules* provide HTTP interfaces that can be used for web-based interaction with eZ publish. While some modules offer an interface to kernel functionality, others are more or less independent of the kernel. The system comes with a collection of modules that cover the needs of typical everyday tasks. For example, the `content` module provides an interface that makes it possible to use a web browser to manage content. It is possible to extend the system by creating custom modules for special needs. Custom modules have to be programmed in PHP. The following table gives an overview of some of the most commonly used modules that come with eZ publish.

Module	Description
`Content`	The `Content` module provides an interface to the content engine in the eZ publish kernel. This module makes it possible to display, edit, search and translate the contents of objects, manage the node tree and so on.
`User`	The `User` module provides an interface to the user management system in the kernel. This module makes it possible to log users in and out of the system. In addition, it provides functionality related to user registration, user activation, password changing, etc.
`Role`	The `Role` module provides an interface to the access control system in the kernel. This module makes it possible to create, modify and delete roles and policies. In addition, it provides functionality for assigning roles to different users and user groups.

Refer to the *Modules* section of the appendix for a comprehensive list of all the built-in modules.

Module execution

Every time an eZ publish site is accessed using a web browser, the client application indirectly interacts with one of the modules present in the system. The requested URL tells eZ publish which module it should execute in order to process the request. In particular, the first part of the URL reveals the name of the module. This is usually the part that comes directly after *index.php* (unless the URI access method is used). The following example shows a typical eZ publish URL:

```
http://www.example.com/index.php/content/edit/13/3
```

This URL’s request is directed at the content module. Another typical example of an eZ publish URL is:

```
http://www.example.com/index.php/user/login
```

By looking at the URL, we see that eZ publish will attempt to execute the user module when processing this request. Obviously, some additional information is also specified in the URLs. In the first example, the name of the module is followed by `"/edit/13/3"`. In the second example, the name of the module is followed by `"/login"`. These additional strings control the behavior of the requested module and are explained below.

Module views

A module consists of a set of *views*. A view can be thought of as an interface to a module. By using views, it is possible to reach various functions that a module provides. For example, among other things, the `content` module provides views for displaying, editing, searching and translating the contents of objects. The name of the view to be accessed appears after the name of the module (separated by a slash) in the URL. In the first example above, eZ publish is instructed to access the `edit` view of the content module. In the second example, eZ publish is instructed to access the `login` view of the user module.

When a view is called, eZ publish starts up the program code associated with that view. Upon completion, the view returns a result to the module, which in turn returns it to the rest of the system. The result is put inside a template variable called `$module_result.content` which is available from the main template (the *"pagelayout"*). Refer to the *Template generation* section of the *Templates* chapter for more information.

View parameters

Some views can have one or more parameters. A view parameter makes it possible to pass information to the view itself and thus allows the view to be controlled from within the requested URL. The view parameters are appended after the name of the view in the URL. In the first example above, the following parameters are passed to the view: "13" and "3". These parameters will instruct the `edit` view of the `content` module to provide an interface for editing the third version of the thirteenth content object in the system. The URL given in the second example does not make use of any view parameters. The view mechanism supports two types of parameters:

- Ordered parameters
- Unordered parameters

Ordered parameters must be separated by slashes and must come after the name of the view. In addition, they have to be provided in the order specified in the module's definition. For example, if the view parameters in the first example get mixed up, eZ publish will attempt to edit the thirteenth version of the third object (instead of the third version of the thirteenth object).

As the name suggests, *unordered parameters* can be provided in an arbitrary order. If the view supports ordered parameters, the unordered parameters must come after the ordered parameters. If the view doesn't support ordered parameters, the unordered parameters will come directly after the name of the view in the URL. The unordered parameters must be provided in pairs. A pair consists of the parameters' names and values separated by slashes. The following example shows an imaginary eZ publish URL with unordered parameters passed to the requested view:

```
http://www.example.com/index.php/
        video/dvd/button/play
```

The address in the example above tells eZ publish to run the imaginary `video` module and execute the `dvd` view. A variable called `button` will be created and made available for the view code. The value of the variable will be set to `play`. It is up to the PHP code of the view to discover this variable and to carry out the necessary sequence of actions.

POST variables

Some views make use of parameters that are submitted using forms through the HTTP `POST` method. For example, the `action` view of the `content` module makes extensive use of `POST` variables.

The default request

In order to produce proper output, eZ publish must know which module it should run and which view should be executed. In other words, every URL has to contain at least the name of an existing module and a view. If an incomplete or mistyped URL is provided, eZ publish will display an error page revealing what's wrong (missing / mistyped module or view). If the requested URL doesn't contain anything after *`index.php`* (except a slash), the default module / view combination will be executed. The default module / view combination can be configured using the `IndexPage` setting under `[SiteSettings]` in an override for *`site.ini`*. The default value is `"/content/view/full/2"`. It instructs eZ publish to show a full view of node 2, the content top-level node. In other words, if the following request is made:

```
http://www.example.com/index.php
```

...eZ publish will behave as if the following URL was requested:

```
http://www.example.com/index.php/
        content/view/full/2
```

No redirection or page reload will be made, which means that the address field of the browser will remain unchanged.

URL translation

This section explains the different URL types which can be used with eZ publish, and how the URL translator works. By default, eZ publish is capable of handling two types of URLs:

- System URLs
- Virtual URLs

System URLs

A *system URL* tells eZ publish which module should be run and which view should be executed. It may contain additional parameters which are passed to the target view. Every system URL follows the same structure and can be broken down into the following components:

- Module name
- View name
- View parameters

The view parameters are optional and may consist of ordered and / or unordered values. A comprehensive description of view parameters can be found in the *Modules and views* section. The following model shows the required sequence of the different URL components:

```
http://www.example.com/index.php/module/view/
ordered_view_parameters/unordered_view_parameters
```

URL component	Description
Module	The name of the module that should be run.
View	The name of the view that should be executed.
Ordered view parameters	Some views make use of ordered parameters.
Unordered view parameters	Some views make use of unordered parameters.

The following example shows a typical system URL:

```
http://www.example.com/index.php/content/edit/13/3
```

By looking at the URL, we can tell that it will instruct eZ publish to run the `content` module and execute the `edit` view. The values *"13"* and *"3"* are parameters that will be passed to the view itself. Note that the exact style of the URLs depend on the access method used and the way the web server is configured. For example, the web server may be configured to hide the *index.php* part of the address.

Virtual URLs

A *virtual URL* (also known as *URL alias or nice URL*) is an alias for an existing system URL. Virtual URLs are easier to remember and sometimes shorter than system URLs. While system URLs reveal a great deal about what eZ publish is instructed to do, virtual URLs do not reveal system-specific information. In other words, virtual URLs cannot be broken down to components in the same way as system URLs. The following example shows a typical virtual URL:

```
http://www.example.com/company/about
```

There are two types of virtual URLs: ones that are automatically generated and maintained by eZ publish and ones that are created and maintained by the site administrator. However, all virtual URLs are treated equally and handled in the same way. The system keeps track of the URLs in a table which consists of two columns: virtual / source address and system / destination address. An entry in the URL table might look something like:

Virtual / source address	System / destination address
company/about	content/view/full/1885

An example of a URL using the virtual address similar to that shown in the table above:

```
http://www.example.com/company/about
```

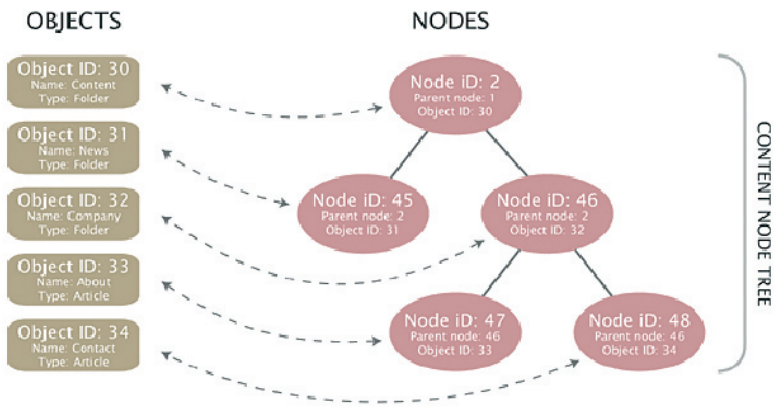
According to the table above, the virtual URL will be translated internally to the following system URL:

```
http://www.example.com/content/view/full/1885
```

Both URLs are valid and will produce the same output, in this case a full view of node number 1885. When the virtual URL is used, the redirection will be done internally, and thus the user will reach the target node without any glitches like redirections, page reloads, etc.

Automated virtual URL generation and maintenance

Every time an object is published, the system will automatically generate a virtual URL for each of the object's node assignments. The generated URL for a node is based on the node's location in the tree and the name of the object the node encapsulates. The virtual URLs generated for the nodes are handled completely by the system and cannot be changed using the Administration Interface. The following illustration shows an example of objects, nodes and a corresponding URL table.



URL TABLE

Source/virtual address	System/destination address
news	content/view/full/45
company	content/view/full/46
company/about	content/view/full/47
company/contact	content/view/full/48

Figure 2.23. Objects, nodes and the URL table

The example above demonstrates how virtual URLs are generated. For each node, the system generates a path of strings separated by slashes. The strings in the path are the names of the objects referenced by the nodes up to and including the target node. Special symbols are converted to underscores and special characters are converted using the built-in transliteration feature. For example, the Norwegian characters "æ", "ø" and "å" are converted to "ae", "oe" and "aa". If the system is about to generate a virtual URL that already exists, it will simply append an underscore and a number at the end of the newly generated address to eliminate the risk of duplicate URLs.

When the name of an object is changed, the system will change the virtual URLs for the involved nodes. In addition, an internal redirection will be created, which will ensure that the old URL still works. The old virtual URL will keep working until the same URL needs to be generated for a node. In this case, the old virtual URL will be deleted.

Manual virtual URLs and translations

It is possible to manually add, edit and remove virtual URLs using the Administration Interface. The URL translator mechanism makes it possible to add three types of translations:

- New virtual URL for an existing system URL
- Secondary / alternative virtual URL for an existing virtual URL
- Wildcard-based URL forwarding

URL handling

When eZ publish receives a request, it looks at the URL sent by the web browser. The address is stripped of unnecessary parts such as the host and domain name, etc. If the address exists as a virtual URL in the table, eZ publish will attempt to process the corresponding system URL. If the address doesn't exist, eZ publish will attempt to interpret it as a system URL.

Designs

This section explains the concept of designs and how eZ publish handles different *designs*. As mentioned in the beginning of this chapter, design refers to the way content is marked up and visually presented. When talking about a design, we're talking about the things that make up a web interface: HTML, style sheets, images, etc. that are not part of the content. All files related to appearance reside in the `design/` directory. An eZ publish installation is capable of handling a virtually unlimited number of designs. Each design has its own dedicated subdirectory within the main design directory. The name of the subdirectory also functions as the name of the design. A typical eZ publish design consists of the following components:

- CSS files
- Image files
- Font files
- Template files

The siteaccess dictates which design should be used. By making use of different siteaccesses, it is possible to combine different content and designs. A typical eZ publish site consists of two siteaccesses: a public interface for visitors and a restricted interface for administrators. Both siteaccesses use the same content (database and `var/` directory) but they use different designs. The administration siteaccess would most likely use the built-in administration design. The public siteaccess would use a custom design.

Default designs

An eZ publish distribution comes with at least two default designs:

- admin
- standard

The `admin/` directory contains all design-related files that make up the built-in Administration Interface. The `standard/` directory contains a set of standard / default design-related files such as the default / standard templates, images, etc. The contents of these directories should

not be altered. Instead, custom designs should be used when desired. A custom design can be added by creating a new subdirectory within the main *design/* directory.

Design directory structure

All files belonging to a specific design are located inside the directory which corresponds to that design. The name of the directory functions as the name of the design itself. An eZ publish design directory typically contains the following subdirectories:

Subdirectory	Description
fonts/	Font files used by the `texttoimage` template operator which is capable of displaying text using truetype fonts.
images/	Non-content specific images such as banners, logos, graphical layout elements, etc.
override/	Custom templates that will be used instead of the default / standard templates. These files will be triggered by template override rules that are specified in a configuration override for *override.ini*. Refer to *The template override system* section of the *Templates* chapter for more information about this feature.
stylesheets/	CSS files.
templates/	Main template(s), for example the pagelayout, header, footer, etc. and custom templates that will be used instead of the standard / default templates.

Design combinations

A siteaccess may make use of several designs. This means that the final result generated by eZ publish can be a combination of files originating from various designs. A siteaccess is capable of using a combination of the following:

- One main design
- One or more additional designs
- One standard design

A siteaccess should always have at least a main design and a standard design. While the main design can be set to anything, the standard design should not be modified. The default configuration is to use the built-in standard design. It ensures that eZ publish always finds the necessary templates, ensuring that any kind of content can be rendered without problems. A more in-depth explanation is presented below.

Automatic fallback

If eZ publish is unable to find a design-specific file (a stylesheet, a template, an image, etc.) within the main design, it will automatically attempt to locate the file elsewhere. The system will sequentially go through all the additional designs (if specified), looking for the requested file. At last, if the requested file still hasn't been found, eZ publish will attempt to locate the missing file within the standard design. The following diagram illustrates this functionality.

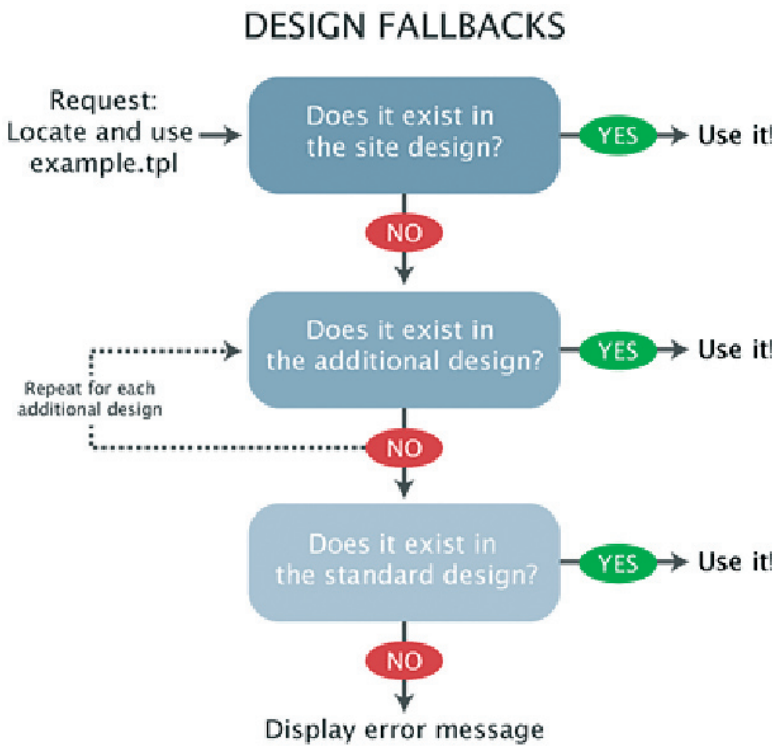


Figure 2.24. The design fallback mechanism

Configuration

The designs that will be used must be specified in the `[DesignSettings]` block within an override for the *`site.ini`* configuration file. The following directives can be used:

- SiteDesign
- AdditionalSiteDesignList
- StandardDesign

The *SiteDesign* directive specifies the main design. The *AdditionalSiteDesignList* directive specifies an array of additional site designs. The *StandardDesign* directive specifies the standard design. Even though it is possible to change the standard fallback design, it is not recommended. The `StandardDesign` directive should always be set to the built-in standard design. This is already defined in the default *`site.ini`* file, and there is no need to set the standard design from within an override. If there is a need for a custom fallback design, it should be specified using the `AdditionalSiteDesignList` directive. The automatic fallback mechanism makes the reuse and combination of designs simple.

Example

The following example shows how to configure the following design settings in an override for the *site.ini* configuration file:

- ”my_design” is the main design
- ”fallback_one” is the first additional design
- ”fallback_two” is the second additional design
- ”standard” is the standard fallback design

```
...
[DesignSettings]
SiteDesign=my_design
AdditionalSiteDesignList[]=fallback_one
AdditionalSiteDesignList[]=fallback_two
StandardDesign=standard
...
```

In this example, if eZ publish is unable to find the requested file within the main ”my_design”, it will automatically fallback to the additional designs. At first, the system will look for the requested file within the ”fallback_one” design directory. If the requested file is not found, the system will look in the ”fallback_two” design directory. If the file still hasn’t been found, the system will attempt to locate it within the ”standard” design directory. The standard directory will most likely contain the requested file (unless a custom template is requested).

Access control

This section explains how eZ publish manages user accounts and access permissions. The system comes with a built-in access control mechanism that can be used to limit access to content or to certain functionality. The access control system is based on the following elements:

- User
- User group
- Policy
- Role

A *user* corresponds to a valid user account on the system. A *user group* consists of users and other user groups. A *policy* is a rule that grants access to content or to a certain system function. (For example, a policy may grant read access to a collection of nodes.) A *role* is a named collection of policies. A role can be assigned to users and user groups. The following sections explain each of these elements in depth.

The following illustration shows the relationships between the elements in the list above.

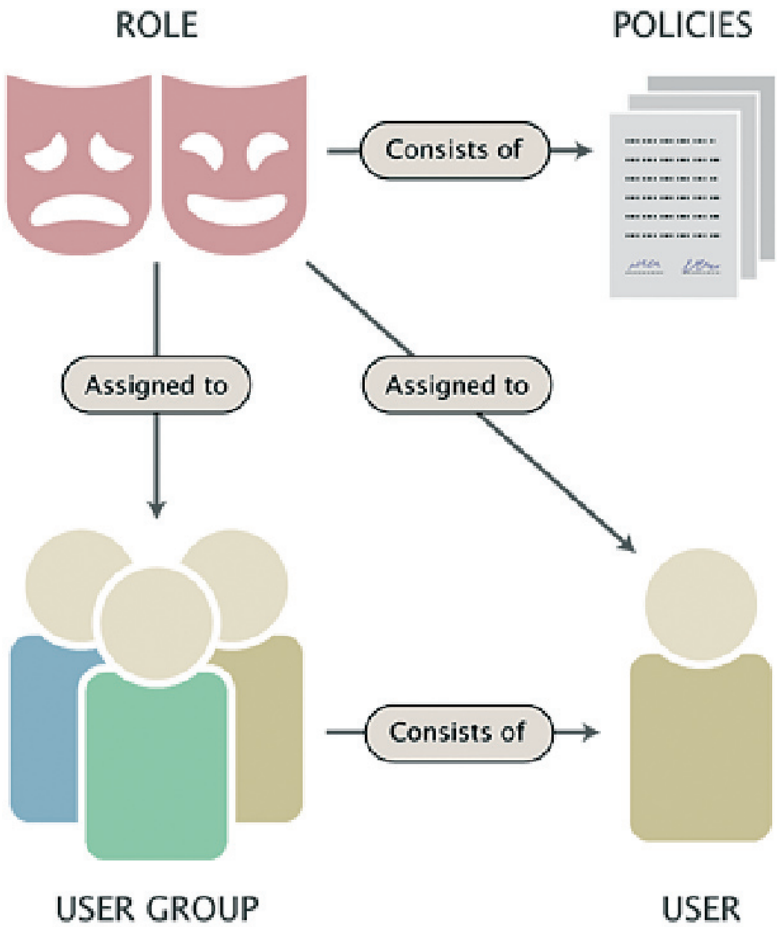


Figure 2.25. Users, groups, policies and roles

User

A *user account* is represented by a content object (with at least one node assignment) that contains information about a specific user. The default `User` class allows the storage of the following elements: first name, last name, email, username and password. The last three elements (e-mail, username and password) are provided by the ”User account” datatype. This is a special datatype which plugs more deeply into the system. Instances of any content class containing the ”User account” datatype will function as valid users on the system. In other words, if there is a need to store additional information about users, it is possible to either modify the default user class or to create a custom class that contains this datatype. A user account can be enabled or disabled from within the Administration Interface. When disabled, the account will continue to exist, but the user will not be able to log in until the account is enabled.

Note that the default configuration does not allow different users to be registered with the same e-mail address. This is a built-in precaution which can be disabled by setting the `RequireUniqueEmail` directive within the `[UserSettings]` block of a configuration override for *`site.ini`* to `FALSE`.

User ID

Every user has a unique identification number which is the same as the ID number of the object that represents the user account. *User ID*s are used by other objects on the system. In particular, an object contains references (user IDs) to the initial creator and to all users who have created versions within that object. Removing a user account might lead to an inconsistent state where objects have owner / modifier references to nonexisting user accounts. Because of this, it is not recommended to remove users from the system; the accounts to be removed should be disabled instead.

User group

A *user group* is a content object (with at least one node assignment) that contains user accounts and other user groups. In other words, a user group is a collection of users (similar to a directory containing files and subdirectories on a filesystem).

Policy

A *policy* is a rule that grants access to a specific function, or all functions, of a module. A policy consists of the following elements:

- Module name
- Function name
- Function limitation

The *module name* indicates the module to which the policy grants access. The *function name* specifies the function to which the policy should be limited. A policy can either be restricted to a single function, or grant access to all functions of a module. A module can have several or no functions. The functions are assigned to the module's views, so the access requirements for a view are controlled by the functions assigned to that view. The functionview assignments cannot be altered from within the Administration Interface. A policy granting access to a module's function can be further restricted using *function limitations*. This can only be done if the function itself supports limitations. A function may support zero, one or several limitations. The following table shows an overview of available function limitations.

Limitation	Description
Class	The *Class* limitation makes it possible to limit a policy to objects of certain types.
Node	The *Node* limitation makes it possible to limit a policy to a specific node.
Owner	The *Owner* limitation makes it possible to limit a policy to objects that are owned by the user who is logged in.
Parent class	The *Parent* class limitation makes it possible to limit a policy based on the type of object referenced by the parent node.

Limitation	Description
Section	The *Section* limitation makes it possible to limit a policy to objects assigned to certain sections.
Status	The *Status* limitation makes it possible to limit a policy to a certain version status (published, archived, etc.).
Subtree	The *Subtree* limitation makes it possible to limit a policy to a certain part of the content node tree.

Role

A *role* is a named collection of policies. A role can be assigned to users and user groups. It is possible to assign a role with additional limitations. The role limitation feature is typically useful in a case where multiple users with similar permissions have to manipulate different parts of the content node tree. Instead of creating a role for each user, the site administrator can create a generic role and assign it with different limitations to the different users or groups. The role limitations will override the limitations of the role's policies. The following table shows an overview of available role limitations.

Limitation	Description
Section	The *Section* limitation makes it possible to limit a role to objects assigned to certain sections.
Subtree	The *Subtree* limitation makes it possible to limit a role to a certain part of the content node tree.

Webshop

This section explains the e-commerce capabilities of eZ publish. The system comes with an integrated shop mechanism that plugs directly into the object / node tree model. Note that the Webshop was significantly improved in eZ publish 3.8. Among other things, a datatype for storing prices in different currencies and advanced VAT handling was added. The improvements made in 3.8 are not covered in the text below. The Webshop functionality is built around the following components:

- Products
- Value Added Taxes (VATs)
- Discount rules
- Wish lists
- Baskets
- Orders

The following illustration shows how the Webshop components interconnect and work together.

Figure 2.26. The integrated e-commerce solution

A product is represented by a content object (with at least one node assignment) that contains information about the product itself, and a price. The price can be affected by a value added tax and / or a discount rule. A discount rule can be configured to reduce the price of certain products by a percentage. The products can be put into a user's wish list and / or shopping basket. A user's wish list and basket can be modified at any time. The contents of the shopping basket can be purchased by initiating the checkout process. Once the checkout process is completed, an order will be created. The system will automatically notify the site administrator and the user who placed the order by sending out e-mails. A list of placed orders and sales statistics can be viewed using the Administration Interface. An order is assigned a status which may be changed by a user with sufficient permissions. A status log is kept for each order.

Value added taxes

The system allows the site administrator to set up different kinds of value added taxes (VATs). A VAT consists of a name and a percentage. The Administration Interface makes it possible to add, remove and modify VATs. The VATs are used by the price datatype.

The price datatype

As described above, a product consists of a content object and a price. The *price* must be represented by an attribute that makes use of the built-in price datatype. This is a special datatype which plugs more deeply into the system. Instances of any class containing the price datatype will automatically be treated as products. The ”Multiprice” datatype (added in 3.8) makes it possible to store / provide prices in multiple currencies for the same product. Note that it is not possible to use the ”Price” and the ”Multiprice” datatypes together on the same site.

A class attribute represented by the price datatype makes use of one of the predefined VATs. There are two ways in which the selected VAT can be used. This configuration depends on how the product prices are entered when the objects are created. The first alternative (**Price inc. VAT**) is to be used if the prices entered already include the value added tax. The second alternative (**Price ex. VAT**) should be used if the prices entered do not contain the value added tax. When the first alternative is used and the product is viewed, the price that was entered will be shown. When the second alternative is used and the product is viewed, the price will be the price that was entered, plus the VAT. When the object is in the basket and the basket is viewed, it is possible to see the price of the products both with and without the VATs (regardless of which approach was used).

Discount rules

The final price of a product can be affected by a *discount rule*. A discount rule can be configured to reduce the prices of certain products by a percentage. The discount rules can be placed in different discount rule groups and are always active (there is no way to turn them on or off). By default, a newly created discount rule affects all the products in the system. However, a discount rule can be easily limited to a group of products. A discount rule can be limited in two ways, which are mutually exclusive. The first alternative is to use a combination of the ”Product type” and ”Section” limitations, which are described in the table below.

Limitation	Description
Product type	The *Product type* limitation makes it possible to limit a discount rule to products of certain types (only classes that make use of the price datatype will be shown). The default setting is **Any**, which means that it will affect all types of product objects.
Section	The *Section* limitation makes it possible to limit a discount rule to products assigned to certain sections. The default setting is **Any**, which means that it will affect product objects in all sections.

The second alternative is to add individual products to the discount rule’s product list. When the individual product list is used, the ”Product type” and ”Section” limitations will be omitted and only the products in the list will be affected.

Shop-related datatypes

The following table shows the datatypes that plug in to the e-commerce subsystem of eZ publish.

Datatype	Description
Price	When used as an attribute in a content class, the *Price* datatype connects the instances (objects) of that class to the Webshop system. As soon as an object has a price attribute, users can put the object in their baskets and / or wish lists. Objects without a price attribute cannot be put into a user's basket and / or wish list.
Option	The *Option* datatype makes it possible to create a single group of options for each content object. Each option can be assigned a short text and an additional price. For example, it can be used to sell T-shirts in different colors, where the price is different for some (or all) colors.
Multi-option	The *Multi-option* datatype makes it possible to create multiple groups of options for each content object. Each option can be assigned a short text and an additional price. This datatype works in the same way as the "Option" datatype. The only difference is that instead of supporting only one group of options, it allows the creation of multiple groups of options for each content object.
Range-option	The *Range-option* datatype makes it possible to create a single group of enumerated options for each content object. For example, it can be used in a scenario where the goal is to sell shoes of different sizes and the size does not affect the price. For each content object, the administrator needs to set up the available range (if applicable).

Workflows

This section explains the workflow capabilities of eZ publish. The system comes with an integrated workflow mechanism that makes it possible to perform different tasks with or without user interaction. The workflow implementation is based on the following components:

- Events
- Workflows
- Workflow groups
- Triggers

An event is the smallest entity of the workflow system. It carries out a specific task. eZ publish comes with a collection of events that cover the needs of typical everyday tasks. For example, the built-in "approve" event makes it possible to have the contents of an object approved by an editor (a user) before it is published. The built-in events are documented in the *Workflow events* section of the online documentation. It is possible to extend the system by creating custom events for special needs. Custom workflow events have to be programmed in PHP. Refer to the *Extensions* chapter for an example of how to create custom workflow events.

The following illustration shows the relationships between the elements in the list above.

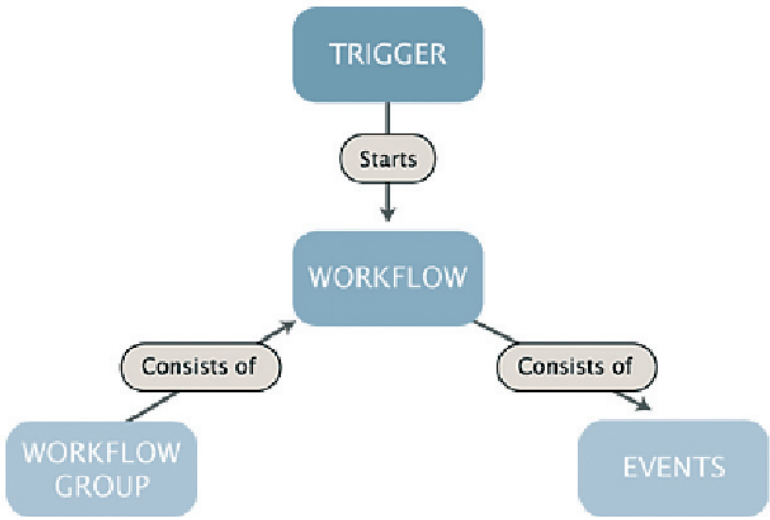
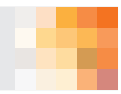


Figure 2.27. The workflow system

A workflow is a collection of events. In other words, it defines an ordered sequence of actions that will be executed when the workflow is running. The workflows can be placed in different groups. A workflow group is a collection of workflows. A workflow is initiated by a trigger. Although a trigger is only capable of initiating a single workflow, several other workflows can be started using the built-in multiplexer event (from within the workflow that was originally initiated by the trigger). A trigger is associated with a function of a module. It will start the specified workflow either before or after that function has been completed. The following table gives an overview of the standard / built-in triggers.

ID	Module	Function	Connection type
1	content	publish	before
2	content	publish	after
3	shop	confirmorder	before
4	shop	checkout	before
5	shop	checkout	after

Chapter 3. Templates



This chapter explains the eZ publish template system. It describes both the template language and the way the system handles the template files. After reading this chapter, you will understand:

- What a template is, and what it is not
- Template types (pagelayout, node and system templates)
- Template structure
- Template language
- The main template (pagelayout)
- Template variables available in the pagelayout
- How basic template tasks are accomplished
- How information can be retrieved from the CMS
- The template override system

Template basics

This section explains the concepts behind templates and the template system. eZ publish uses templates as the fundamental unit of site design. A template is a custom HTML file that describes how some particular type of content should be displayed. A template file always ends with a `.tpl` extension. The built-in / default templates follow the XHTML 1.0 Transitional specification. (Although note that the templates themselves do not validate as XHTML 1.0 documents.) In addition to standard HTML syntax, a template consists of eZ publish-specific code. The eZ publish-specific code makes it possible to extract information from the system and provides common programmatic features like conditional branching, looping, etc. All eZ publish-specific code must be placed inside a set of curly braces: { and }.

The following example shows part of a template that prints out the current time:

```
...
<h1>Time machine</h1>
<p>
    The current time is:
        {currentdate()|l10n( time )}
</p>
...
```

The example above demonstrates how standard HTML is mixed with eZ template language code. It shows the usage of the `currentdate` and the `l10n` template operators. (Unlike functions, operators can receive input through a pipe). Since `currentdate` returns a UNIX timestamp, it must be formatted using the l10n localization operator (or the output would not make sense to humans). This is done by directing the output of `currentdate()` into `l10n` using a pipe: |. The output will be the requested information formatted according to current locale settings. The *`time`* parameter tells the operator to output only the time; it could also have outputted *`date`*, *`shortdate`*, *`datetime`* and so on.

Template generation

The template system is component-based. In most cases, a single page of HTML generated by eZ publish is created by combining several templates. At a minimum, eZ publish always renders the main template, which is called *pagelayout*. The pagelayout contains the `html`, `head` and `body` tags; together with the stylesheets it dictates the overall look of a site. Among other things, it describes the visual structure (layout, title, logo, menu, footer, etc.) that will be presented for each web page the system generates.

Each incoming request tells eZ publish to run a specific module and to execute one of the module's views. When finished, the requested module / view combination will generate a result by making use of a template. The result can be accessed through the `$module_result` array which is available in the pagelayout template. The following illustration shows a simplified three-step explanation of how eZ publish responds to HTTP requests.

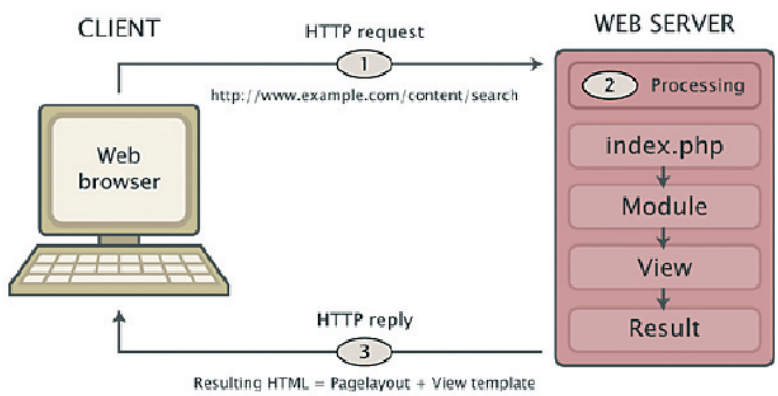


Figure 3.1. Client - server cycle

Templates used by views are often referred to as "view templates". Whenever a view has finished running, it will issue an internal template request. The requested template will be interpreted, processed and converted to HTML. After processing, the system will put the resulting HTML in the module's result array. It can be accessed like this: `{$module_result.content}`. When accessed, the system simply prints out its contents. This makes it possible to include the HTML code that is generated by the views in the pagelayout. The following illustration shows how the module / view result (generated by different modules / views, depending on the request) is included in the pagelayout:

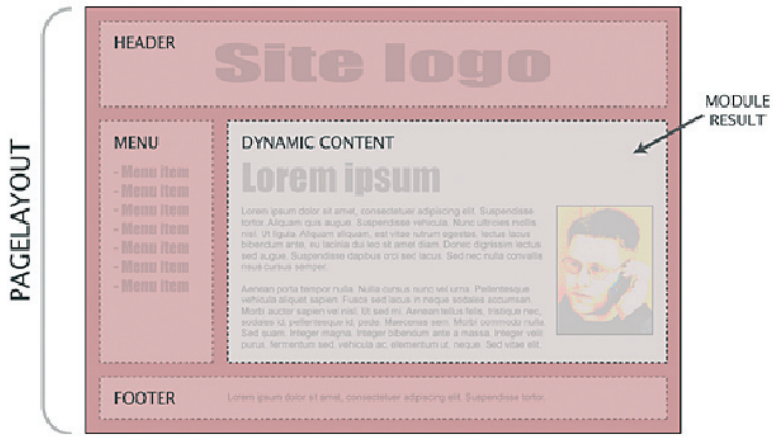


Figure 3.2. The module result as a part of the pagelayout

View templates

There are two types of view templates: node templates and system templates. Node templates are only used when a node is viewed (for example when a system URL starting with ”/content/view” or the virtual URL of a node is requested). System templates typically provide HTML interfaces to specific eZ publish features. For example, the template used by the `search` view of the `content` module provides an HTML front-end for the built-in search engine.

The difference between the template types described above is the available variables and the combination of override rules that can be used. A node template gives access to a variable (`$node`) that contains information about the node being viewed. Depending on the view that was called, a system template typically gives access to several variables. A template override rule makes it possible to display custom templates in specific cases. The override rules for node templates are much more flexible than the override rules for system templates. For example, it is possible to set up complex rule combinations that depend on the type of node being viewed, the depth of the node in the tree, the section to which the node’s object is assigned and so on. Refer to *The template override system* section for a detailed description of the template override mechanism.

Node templates

Whenever eZ publish receives a request to output information about a node (either from a system URL or a virtual URL), it executes the `view` view of the `content` module. If a system URL is used, both the desired view mode and the target node must be specified in the URL. If a virtual URL is used, eZ publish will determine which node should be accessed by looking up the corresponding system URL in the internal URL table. When a virtual URL is used, the system will always use the full view mode.

The templates for the different view modes must be placed inside the *`templates/ node/ view/`* directory of a design. If the requested file is not found within the main design of the siteaccess, the system will search for it in the additional designs and the standard design. Re-

fer to the *Automatic fallback* section of the *Concepts and basics* chapter for more information about this feature. The `templates/node/view/` directory of the standard design contains templates for different view modes. A basic custom design typically contains a pagelayout and a full view template. The following illustration shows the locations of these templates in a custom design called ”example”.

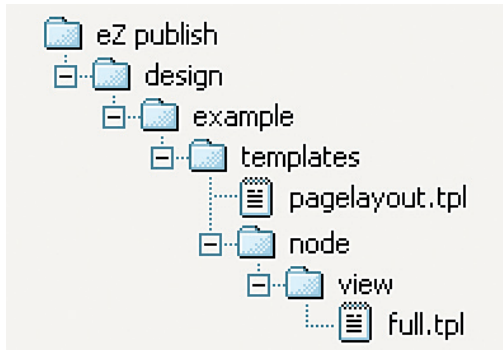


Figure 3.3. Location of pagelayout and full view template in example design

When a node is requested (and there are no template override rules for node templates), eZ publish will generate a page using the following templates:

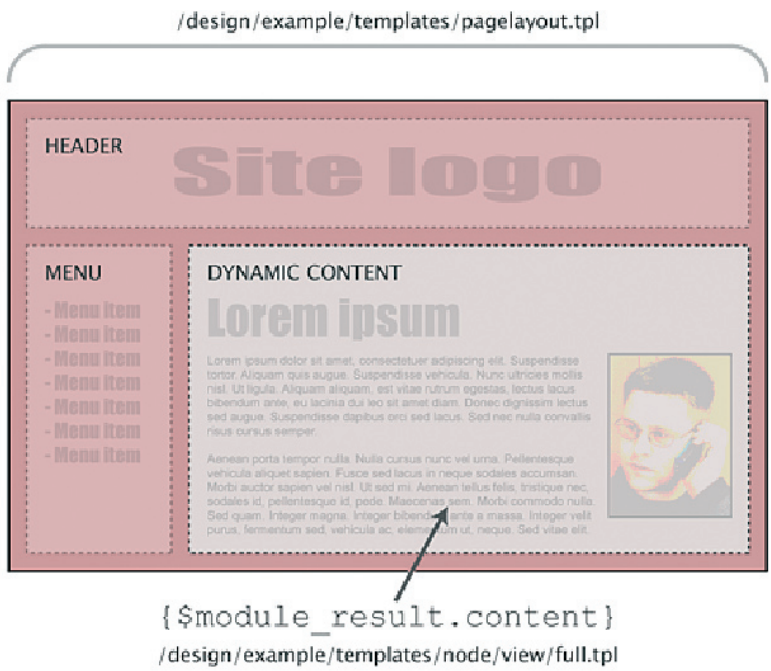


Figure 3.4. Pagelayout + node view full template

Custom node templates

Typical eZ publish sites use custom node templates. This is because there is almost always a need to display various types of nodes in different ways. For example, information pages need to look different than news articles; the welcome page has to be formatted in a special way; and so on. Unlike custom system templates (which are mostly just modified copies of the standard templates placed in a custom design), custom node templates are created as override templates. The override templates are triggered by the template override system. This system offers a flexible mechanism that can be programmed to use different templates based on various conditions. For example, it can be programmed to use a template called *`article.tpl`* when the system needs to show the contents of nodes referencing article objects and at the same time use the template *`special_article.tpl`* when a specific article is accessed. Refer to the *Template override system* section for more information about how this mechanism works.

The $node variable

Whenever the system makes use of a node template, a variable called `$node` will be available in the template. This variable is automatically set by the system, and it contains an `ezcontentobjecttreenode` object that represents the requested node. This variable is used to extract and display information about the node and the object it encapsulates. Refer to *Outputting node and object data* for information about how to display node / object data.

System templates

Whenever eZ publish receives a request to do something other than display a node (in other words, the URL does not contain "/content/view" or isn't the virtual URL of a node), it will use a system template. There are two main differences between system templates and node templates:

- System templates provide access to various variables (depending on the view that was requested). A node template only provides access to a `$node` variable representing the requested node.
- The override rules for node templates are much more flexible than the override rules for system templates.

An eZ publish distribution provides default templates for all views. These templates are located in the *`templates/`* directory of the standard design. A view typically uses a template located in a subdirectory with the same name as the module to which the view belongs. The name of the template is usually the same as the name of the view (with a ".tpl" extension). For example, the `login` view of the `user` module corresponds to a template called *`login.tpl`* inside a directory called *`user/`*. Another example is the `basket` view of the `shop` module. This view will use a template called *`basket.tpl`* within the *`shop/`* directory.

Custom system templates

Although eZ publish provides all the necessary system templates (contained in the standard design), a typical eZ publish site makes use of customized system templates. This is because the default templates usually need to be tailored to the style of a custom design. Unlike custom node templates, which are mostly provided using the template override system, custom system

templates are usually modified copies of the standard templates located in the custom design. For example, a custom template for the `login` view of the `user` module in a design called ”example” would be *`design/example/ templates/user/login.tpl`*. A custom template for the `search` view of the `content` module would be *`design/example/templates/content/search.tpl`*.

Design combinations

As mentioned above, a custom design typically contains a set of customized system templates. However, creating a custom design that provides templates for all possible scenarios would be unnecessary work. This is why the standard design should always be used as the default. The automatic fallback system makes it possible to combine several designs so that the main design (which is usually a custom design) does not have to provide all the necessary templates. Whenever eZ publish is unable to find a template within the main design of the siteaccess, the system will look for it in the additional designs and the standard design.

Commonly used system templates

The following table shows some of the most commonly used system templates.

Request	URL	Module	View	Template
Search interface	/content/search	content	search	/templates/content/search.tpl
Shopping basket	/shop/basket	shop	basket	/templates/shop/basket.tpl
Login page	/user/login	user	login	/templates/user/login.tpl
User registration	/user/register	user	register	/templates/user/register.tpl

The pagelayout

The pagelayout is the main template. It dictates the overall look of a site. The filename of the pagelayout template must be *pagelayout.tpl*. It has to be placed inside the *templates/* directory of a design. If eZ publish is unable to find a pagelayout within the current design (specified by the siteaccess), it will attempt to use the pagelayout template provided by one of the fallback designs. The following illustration shows the location of the pagelayout template located in a design called ”example”.

Figure 3.5. The location of the pagelayout (main) template

The pagelayout contains the `html`, `head` and `body` tags (the outer HTML framework); together with the stylesheets it dictates the overall look of a site. Among other things, it describes the visual structure (layout, title, logo, menu, footer, etc.) that will be presented for every page request. The following example shows a basic pagelayout:

```
<!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>

<style type="text/css">
    @import url({'stylesheets/core.css'
        |ezdesign});
    @import url({'stylesheets/debug.css'
        |ezdesign});
</style>

{include uri='design:page_head.tpl'}

</head>

<body>
{$module_result.content}

<!--DEBUG_REPORT-->

</body>
</html>
```

The document type

The first line in the pagelayout is used to declare the document type of the pages generated by eZ publish. Per HTML and XHTML standards, a `DOCTYPE` (short for "document type declaration") informs browsers and syntax validation engines about the version of (X)HTML that is being used. (This information should be included at the very top of every web page, which is why it is the first part of the pagelayout.)

The `DOCTYPE` declaration is a key component for rendering and page compliancy. A `DOCTYPE` that includes a full URL tells the browser to render the page in standards-compliant mode, treating the (X)HTML, CSS, and DOM structures as they should be treated according to the standards. A missing, incomplete or outdated `DOCTYPE` throws most browsers into *Quirks* mode. In this mode, the browser assumes that the document was written using old-fashioned, invalid markup and code per the chaotic industry norms of the late 1990s. In other words, the page will most likely not render according to standards and it will certainly not validate.

The HTML tag

HTML tags encapsulate the marked-up contents of the web page. In addition to the tag itself, the HTML tag in the example above includes the URL for the XHTML specification. XHTML is a family of current and future document types and modules that reproduce, subset, and extend HTML 4. The XHTML family document types are based on XML, which means that they are designed to work in conjunction with XML-based user agents. Another way of describing them is to say that they are XML documents that can also be parsed by web browsers.

In document processing, it is often useful to identify the natural or formal language in which the content is written. The `lang` and `xml:lang` attributes specify the language of the entire XHTML element. The value of the `xml:lang` attribute takes precedence. The language values should be set to the language used throughout the site. The values of the attributes are language identifiers defined by ISO 3166-1 (and the corresponding ISO 3166-1-alpha-2) standards.

The head tag

The `head` tag contains information about the document itself. The information contained here does not display on the page in the web browser. Only the contents of the `title` tag will be made visible (as the title of the browser window). The `head` tag contains meta-data for the HTML document, such as information about which CSS files should be used, a description of the document itself, keywords and so on.

Cascading Style Sheets

The pagelayout in the example above makes use of two CSS files: *`core.css`* and *`debug.css`*. The code encapsulated by curly braces is eZ publish-specific code. The text within the quotes is being piped into a template operator called `ezdesign`. The operator prepends the text with the path to the current design directory (the one that is specified using the `SiteDesign` configuration directive). This technique ensures that the path to the CSS files is always correct, regardless of the access method. For example, if the name of the current design is ”my_design” and it includes a CSS file called *`example.css`*, the following output will be produced:

```
@import url
    (”/design/my_design/stylesheets/example.css”);
```

The *`core.css`* and *`debug.css`* files are part of the standard design that comes with eZ publish. It is not necessary to have these CSS files in the *`stylesheets/`* directory of a custom design. If eZ publish is unable to find the files within the current design, it will automatically use the ones in the standard design. Refer to the A*utomatic fallback* section of the *Concepts and basics* chapter for a detailed description of the fallback mechanism. Because of the fallback system, the style part of the pagelayout above will most likely result in the following HTML output:

```
...
<style type=”text/css”>
     @import url
         (”/design/standard/stylesheets/core.css”);
     @import url
```

```
        ("/design/standard/stylesheets/debug.css");
</style>
...
```

The core stylesheet

The *core.css* file defines a standard set of basic styles (font styles, sizes, margins, etc.) for both general HTML elements and some eZ publish-specific classes. The eZ publish-specific classes are used by the standard templates. A site that makes extensive use of the default templates should always have the *core.css* file included in the pagelayout. Otherwise, the missing styles may cause unexpected rendering of various elements.

The standard *core.css* file should never be changed. If there are basic styles in *core.css* that don't fit the visual environment of a site, a modified version of *core.css* may be placed in the custom design that the site uses. However, the recommended solution is to create a completely new CSS file that contains both custom classes and overrides for elements defined in *core.css*.

The debug stylesheet

The *debug.css* file contains styles used to format the debug output which appears at the bottom of the page when debug output is enabled. Use of the *debug.css* file is only necessary during the development of the site (which is typically when debug information is needed). Therefore it can be removed or commented out before the site is launched.

Document information

The system is capable of automatically generating information about the page itself (title, meta tags, keywords, etc.). This is done via the inclusion of the page head template (*page_head.tpl*), which is located in the templates directory of the standard design. If eZ publish is unable to find the requested file in the current or custom design, it will automatically fallback and use the file located in the standard design.

The body tag

The `body` tag defines the document's body, which contains the contents of the web page (text, images, etc.), marked up in an orderly fashion. At a minimum, an eZ publish pagelayout should contain the result from the requested modules. If it doesn't, no dynamic output will be displayed. In other words, the `$module_result.content` should always be part of the main template.

Module result

Upon every request, eZ publish automatically generates an array called `module_result`. This array is available only in the pagelayout template. It contains all necessary information about which module was run, which view was called, the output that was produced and so on. The actual output (for example, the contents of a news article) can be included in the pagelayout by accessing the "content" element of the `$module_result` array. The syntax is:

```
{$module_result.content}
```

When the pagelayout is rendered, the `{$module_result.content}` will be replaced with the actual output produced by the requested module. Refer to the *Variables in pagelayout* section for an overview of the template variables that can be accessed from within the pagelayout.

Debug information

The last part of a typical eZ publish pagelayout is an HTML comment that looks like this:

```
<!--DEBUG_REPORT-->
```

If the debug information is turned on, eZ publish will replace this comment with a debug report when the pagelayout is processed. In other words, the debug report will be included as part of the generated page; it will not cause invalid output by breaking the HTML structure. The debug reports generated by eZ publish are valid XHTML 1.0 Transitional.

The page head

The standard design contains a page head template that can be used to automatically generate tags that should be included in the head section of every HTML response. The output of the standard head template (*`design/ standard/template/page_head.tpl`*) can be broken down into the following group of tags:

- Title tag
- Meta tags
- Link tags

The following HTML dump shows an example of the output from the standard page head template.

```
<title>Current / Parent /
        Top - Site name</title>

<meta http-equiv="Content-Type"
        content="text/html;
        charset=iso-8859-1" />
<meta http-equiv="Content-language"
        content="eng-GB" />
<meta name="author" content="eZ systems" />
<meta name="copyright" content="eZ systems" />
<meta name="description" content="Content
        Management System" />
<meta name="keywords" content="cms, publish,
        e-commerce, content management, development
        framework" />
<meta name="MSSmartTagsPreventParsing"
        content="TRUE" />
<meta name="generator" content="eZ publish" />
```

```
<link rel="Home" href="/" title="Front page" />
<link rel="Index" href="/" />
<link rel="Top" href="/" title="Current /
        Parent / Top - Site name" />
<link rel="Search" href="/content/advancedsearch"
      title="Search Site name" />
<link rel="Shortcut icon"
        href="/design/standard/images/favicon.ico"
        type="image/x-icon" />
<link rel="Copyright" href="/ezinfo/copyright" />
<link rel="Author" href="/ezinfo/about" />
<link rel="Alternate"
        href="/layout/set/print/content/
                view/full/64"
        media="print" title="Printable version" />
```

Title

The contents of the `title` tag are based on the location being viewed (the location within either the content node tree or the system itself) and on the name of the site. The path to the element being viewed is reversed, so the current element becomes the first component of the title. The components of the path are separated by frontslashes. When a node is viewed, the path elements will be the names of the objects which are encapsulated by the nodes that make up the path, up to and including the target node. When a system function is being accessed (for example the login view of the user module: `"/user/login"`), the path will most likely be a reversed version of the module / view combination that was used. The name of the site is appended at the end of the path, separated by a dash. The site name can be configured using the `SiteName` directive in a configuration override for *`site.ini`*.

The example above demonstrates the output of the page head template when a node is being viewed. The name of the object encapsulating the node is ”Current”. The name of the other objects (encapsulated by the parent node and so on) are ”Parent” and ”Top”. The name of the site is ”Site name”.

Meta tags

In addition to the content displayed on a web page, the HTML of the page may also include information about the document itself. This is achieved by using *meta tags*. The information provided within meta tags is usually not visible when the web page is viewed. However, the meta tags are used by the web browser and by some search engines that index and rank the contents of web pages. (Note that meta tags can do many things, including simulation of HTTP features). The standard page head template outputs the most commonly used meta tags. It can be broken down into three types of tags:

- HTTP-EQUIV meta tags
- Generic meta tags
- Additional meta tags

HTTP-EQUIV meta tags

Meta tags with an `HTTP-EQUIV` attribute are equivalent to HTTP headers. These tags control the way a browser interprets the document. Tags using this form should have an equivalent effect when specified as an HTTP header. Some web servers automatically translate the contents of these tags to actual HTTP headers. The `HTTP-EQUIV` meta tags in the page head ensure that the browser (and also search engines) know which character set and language the document uses. The language and character set values are automatically set by eZ publish based on the language and character set used by the site.

Generic meta tags

The generic meta tags make it possible to pass meta information about the document itself. Although the specification of meta tags does not define a set of ”legal” meta data properties, it is a common practice to include generic information such as the name of the author, description of the site, copyright notices, keywords, etc. By making use of the `MetaDataArray[...]` directive in a configuration override for *`site.ini`*, the site administrator can set up a custom set of generic meta tags. eZ publish will generate a list of meta tags from the array of tag names and values. The example above shows the default meta tags that will be used if no custom meta tag configuration is present.

Additional meta tags

The last two meta tags set by the standard page head template prevent the use of smart tags, and display the name of the software used to generate the output. (Smart Tags were a proposed feature of Windows XP that would allow Microsoft and its partners to insert their own links into any web page when viewed with Internet Explorer.)

Link tags

Link tags in the HTML head make it possible to relate the document to other documents. This is done using the `REL` and `REV` attributes. While `REL` links are used to establish relationships, `REV` links are used to establish reverse relationships. Some browsers use the link tags to produce a navigation bar for the site. The link tags generated by eZ publish are specified in the *`link.tpl`* file within the templates directory of the standard design. The standard page head makes use of the *`links.tpl`* file. The default output of the standard page head template produces a basic set of links that can be used to navigate to different parts of the site. The following list shows the link tags that the page head generates:

Link	Description
Home	The ”Home” link points to the root / start page of the site (same as ”Index” and ”Top”). It will always bring the user back to the front page (for example, `http://www.example.com`).
Index	The ”Index” link points to the root / start of the site (same as ”Home” and ”Top”). It will always bring the user back to the front page (for example, `http://www.example.com`).

Link	Description
Top	The ”Top” link points to the root / start of the site (same as ”Home” and ”Index”). It will always bring the user back to the front page (for example `http://www.example.com`).
Search	The ”Search” link points to the `advanced search` view of the `content` module. It will bring the user to the advanced search interface (`http://www.example.com/content/advanced search`).
Shortcut icon	The ”Shortcut icon” defines the location of the favorite / shortcut icon. Most browsers will display this icon in front of URLs in the address field and in the bookmark list. The default shortcut icon is the double square white-orange eZ systems logo. It can be easily replaced by putting a 16x16 pixel icon file (16 color BMP/Windows Icon Format) called *`favicon.ico`* in the *`images/`* folder of a site design.
Copyright	The ”Copyright” link points to the `copyright` view of the `ezinfo` module. The default copyright page of eZ publish will be displayed (`http://www.example.com/ezinfo/copyright`).
Author	The ”Author” link points to the `about` view of the `ezinfo` module. The default about page of eZ publish will be displayed (`http://www.example.com/ezinfo/about`).
Alternate	The ”Alternate” link points to an alternate, printerfriendly version of the page that excludes the elements not needed for a printed document (such as the navigation bar). The printer-friendly version of a page is achieved by using the `set` view of the `layout` module.

Link parameters

The links can be turned off by passing `enable_link=false()` when including the page head template:

```
{include uri='design:page_head.tpl'
         enable_link=false()}
```

The link to the print layout can be turned off by passing `enable_print=false()` when including the page head template:

```
{include uri='design:page_head.tpl'
         enable_print=false()}
```

Variables in pagelayout

The pagelayout template contains variables that can be used to display information about the state of the system and / or to control the output. The following table shows the available variables.

Variable	Type	Description
`$access_type`	array	The name of the siteaccess (as ”name”) and the ID number (as ”type”) of the access method used (1=URL, 2=Host, 3=Port).
`$anonymous_user_id`	integer	The ID number of the content object that represents the anonymous user account (the default / standard value is 10).
`$current_user`	object	The ID number of the ezuser object of the user who is currently logged in. If no user is logged in, the anonymous user account will be used.
`$ezinfo`	array	An array of three strings: ”version”, ”version_alias” and ”revision”. These strings contain information about the eZ publish release being used.
`$module_result`	array	Contains information about the result (and the result itself) generated by the module / view that was executed.
`$navigation_part`	array	A hash containing the name and the identifier (the keys are ”name” and `"identifier"`) of the current navigation part, for example: `"Content structure"` and `"ezcontentnavigationpart"`. The navigation part is used by the Administration Interface to determine which part the user interacts with.
`$requested_uri_string`	string	Contains the site-specific part of the requested URL, for example `"content/view/full/44"` (system URL) or `"company/about"` (virtual URL).
`$site`	array	Contains information about the siteaccess being used (site name, design resource, meta tags, etc.)
`$ui_component`	string	Contains the user interface component which eZ publish uses while the current page is being shown. This variable is used by the Administration Interface.
`$ui_context`	string	Contains the user interface context of eZ publish while the current page is being shown. This variable is used by the Administration Interface to distinguish between different modes (for example, ”navigation”, ”edit”, ”browse”, etc.).
`$uri_string`	string	Contains the system version of the requested URL (for example, `"content/ view/full/13"`).
`$warning_list`	array	Contains an array of warnings related to problems that were discovered when the page was rendered.

$module_result

The `$module_result` array contains the result that was generated by the module and view that were executed. If eZ publish was instructed to display the contents of a node, the variable will contain additional information about the node that was requested. If eZ publish was instructed to do something else (almost anything that is not an actual node view), the result will not contain additional information. The following tables show the contents of the `$module_result` variable in the different scenarios.

The default $module_result

Element	Type	Description
`content`	string	Contains the actual content (the result of templates) generated by the requested view.
`path`	array	An array of hashes containing information about the path which leads to the page currently being viewed. Each hash contains the following keys: `"text"` and `"url"`. The `"text"` element usually contains the name of the module or view (for example, `"Collected information"`). The `"url"` element contains the address. The `"url"` key of the last element in the array is usually set to `FALSE`. This is done in order to prevent linking to the page currently being viewed. The standard page head template uses the path array to build the `title` component of the head section. The path array can also be used to build breadcrumb-style navigation aids (i.e., a set of links that show a page-by-page path from the index page of a site to the currently displayed page.)
`is_default_navigation_part`	boolean	Returns `TRUE` if the default navigation part is being used (the one that is set in the PHP code). Returns `FALSE` if the navigation part of the current module and view has been reconfigured by the site administrator. This can be done by making use of the `Navigation-Part` directive of the `[ModuleSettings]` section within a configuration override for *`module.ini`*.
`navigation_part`	string	The identifier of the current navigation part (for example, `"ezcontentnavigationpart"`). This variable is used by the Administration Interface to determine the navigation part the user interacts with.
`ui_context`	string	The user interface context that eZ publish is in while the current page is being shown. This variable is used by the Administration Interface to distinguish between different modes (navigation, edit, browse, etc.)
`ui_component`	string	The user interface component which eZ publish uses while the current page is being shown. This variable is used by the Administration Interface.
`uri`	string	Contains the site-specific part of the requested URL, for example `"content/view/full/44"` (system URL) or `"company/about"` (virtual URL). This tells eZ publish which module and view should be run.

The $module_result when a node is being viewed

<table>
<tr><th>Element</th><th>Type</th><th>Description</th></tr>
<tr><td>content</td><td>string</td><td>The content (the result of templates) generated by the requested view.</td></tr>
<tr><td>view_parameters</td><td>array</td><td>An array of the parameters that were sent to the view (for example, ”limit”, ”offset”, etc.).</td></tr>
<tr><td>path</td><td>array</td><td>An array of hashes containing information about the path of nodes that lead to the node currently being viewed. Each hash contains the following components:
<table>
<tr><th>Key</th><th>Description</th></tr>
<tr><td>text</td><td>The name of the object referenced by the node.</td></tr>
<tr><td>url</td><td>The system URL of the node (for example "content/view/full/44").</td></tr>
<tr><td>url_alias</td><td>The virtual URL of the node (for example "company/about_us").</td></tr>
<tr><td>node_id</td><td>The ID number of the node.</td></tr>
</table>
The node being viewed will have its "url" and "url_alias" components set to FALSE. This is done in order to prevent linking to the page that is currently being viewed. The "node_id" will also not be available. The path array can, for example, be used to build a breadcrumb trail (that is, a path with names (as hyperlinks) of the objects referenced by the nodes that lead to the target/current node).</td></tr>
<tr><td>title_path</td><td>array</td><td>Similar to the "path" array (see above). When a node is being viewed, the standard page head template uses the "title_path" array to build the title component of the head section.</td></tr>
<tr><td>section_id</td><td>string</td><td>The ID number of the section to which the object referenced by the node being viewed belongs.</td></tr>
<tr><td>node_id</td><td>string</td><td>The ID number of the node being viewed.</td></tr>
<tr><td>navigation_part</td><td>string</td><td>The name identifier of the current navigation part (for example, "ezcontentnavigationpart"). This variable is used by the Administration Interface to determine which navigation part the user is interacting with.</td></tr>
<tr><td>content_info</td><td>array</td><td>Contains information about the node being viewed. Refer to the next section for a list of content_info elements.</td></tr>
<tr><td>cache_ttl</td><td>integer</td><td>The TTL (Time To Live) value of the result generated by the module’s view (as seconds). A TTL of minus one means that the view cache should never expire. A TTL of zero means that the result should never be cached.</td></tr>
</table>

Element	Type	Description
is_default_navigation_part	boolean	Returns TRUE if the default navigation part is being used (the one that is set in the PHP code). Returns FALSE if the navigation part of the current module and view has been reconfigured by the site administrator. This can be done by making use of the NavigationPart directive of the [ModuleSettings] section within a configuration override for *module.ini*.
ui_context	string	The user interface context that eZ publish is in while the current page is being shown. This variable is used by the Administration Interface to distinguish between different modes (navigation, edit, browse, etc.)
ui_component	string	The user interface component used by eZ publish while the current page is being shown. This variable is used by the Administration Interface.
uri	string	The site-specific part of the requested URL, for example "content/view/full/44" (system URL) or "company/about" (virtual URL).

The $module_result content_info elements

Element	Type	Description
node_id	string	The ID number of the node.
parent_node_id	string	The ID number of the parent node.
object_id	string	The ID number of the object referenced by the node.
class_id	string	The ID number of the class of which the object is an instance.
class_identifier	string	The identifier of the class of which the object is an instance, for example, "forum_message".
offset	integer	The offset view parameter.
viewmode	string	The view mode used to display the node (for example, "full", "line", etc.).
node_depth	string	The depth of the node in the content tree.
url_alias	string	The virtual URL of the node (for example "company/about_us").
persistent_variable	n/a	A variable set in one of the templates used by the view that was executed. Regardless of the caching mechanisms used, this variable will be available in the pagelayout. The type of the persistent variable depends on the value it contains. If the variable is not set, it will return a boolean FALSE.

Element	Type	Description
`class_group`	array	The ID numbers of the class groups to which the class (which the object being viewed is an instance of) belongs. This variable is connected with a feature that makes it possible to create template overrides based on class groups. By default, the `class_group` always returns a boolean `FALSE` value because the class group override feature is turned off. It can be turned on by setting the `EnableClassGroupOverride` directive in the `[ContentOverrideSettings]` block of a configuration override for *`content.ini`* to `TRUE`.

The template language

The eZ publish template language makes it possible to extract information from the system and to solve common programmatic issues like conditional branching, looping, etc. All eZ publish-specific code must be placed inside a set of curly braces: { and }. A template file is a combination of HTML and eZ publish template code. Note that the templates themselves do not validate as XHTML documents. Everything that is encapsulated by curly braces will be interpreted by the template parser when the template is processed. Everything outside the curly braces will be ignored and will be sent to the browser without any changes.

Curly brace issues

Since curly braces are reserved for defining blocks of eZ publish template code, these characters cannot be used directly in templates. For example, JavaScript code cannot be inserted directly into a template file because it makes extensive use of curly braces. All non-template-specific code or text that uses curly braces must be enclosed in a *literal* section. The contents of a literal section will be ignored by the template parser. The following example demonstrates the usage of the literal tags:

```
...
{literal}
<script language="JavaScript"
        type="text/javascript">
<!--
    window.onload=function()
    {
        document.getElementById
                ( 'sectionName' ).select();
        document.getElementById
                ( 'sectionName' ).focus();
    }
-->
</script>
{/literal}
...
```

Outputting curly braces

It is possible to output curly braces using two template functions called `ldelim` and `rdelim` (short for "left delimiter" and "right delimiter"). The following example demonstrates the use of these functions:

```
...
This is the left curly brace: {ldelim} <br />
This is the right curly brace: {rdelim} <br />
...
```

The following HTML output will be produced:

```
This is the left curly brace: { <br />
This is the right curly brace: } <br />
```

Comments

As in most programming languages, comments can be used to add inline explanations to the code. Template comments are ignored by the parser and will not be displayed in the resulting HTML. They will be completely removed from the output.

There is only one way to add template comments, and that is by encapsulating a block of code by a matching pair of the "{*" and "*}" sequence of characters ("left curly brace + asterisk" and "asterisk + right curly brace"). In other words, a template comment is just like any other template code, except that the curly braces are accompanied by adjacent asterisks. It is possible to comment both single and multiple lines of code. However, nesting comments is not supported (that is, you cannot comment a chunk of code that is already a comment). It is not possible to add comments inside keywords. The following examples demonstrate the use of comments.

Single line comment

```
{* This is a single line comment. *}
```

The example above will not produce any output.

Multi-line comment

```
{* This is a long comment that
    spans across several lines
    within the template file. *}
```

The example above will not produce any output.

Nested comments (illegal)

```
{* {* Nested comments are not supported! *}
This text will be displayed. *}
```

The example above will produce the following output:

```
This text will be displayed.
```

Variable types

The eZ publish template language supports the following variable types:

- Numbers
- Strings
- Booleans
- Arrays
- Objects

While some variable types can be created on the fly, others need to be created using an operator. Types that may be created directly are numbers and strings. Booleans and arrays must be created using operators; objects may be created using miscellaneous functions and operators. In addition to the types listed above, it is also possible to create and use custom types. Custom types must be represented as objects.

Numbers

Numbers are numeric values. Numbers may be integer or floating-point values and may be positive or negative. The values are represented using PHP integers and floats. Refer to the PHP documentation regarding valid ranges and levels of precision. Decimals are always indicated using periods (there is no localization at this level). The following example demonstrates how different numbers can be used directly within template code:

```
{13}
{1986}
{3.1415}
{102.5}
{-1024}
{-273.16}
```

Strings

A *string* is an arbitrary sequence of characters (text) that is enclosed by a matching pair of either single (‘) or double (”) quotes. If the quotes are omitted, the string will be interpreted as a function name. This will most likely result in a template error. Strings are usually expressed in the following way:

```
{‘This is a string.’}
{”This is another string.”}
```

The output of the example above would be:

```
This is a string.
This is another string.
```

Using quotes

It is possible to use quotes inside strings. This can be done by either using a different kind of quote symbol or by making use of the escape character (backslash). The following examples demonstrate the use of quotes inside strings:

```
{‘The following text is double quoted:
        ”Rock and roll!” ‘}
{”The following text is single quoted:
        ‘Rock and roll!’ ”}
{‘Using both single and double quotes:
        ”Rock\’n roll!” ‘}
{‘Using single quotes in a single-quoted string:
        \’Rock\’n roll!\’ ‘}
{”Using double quotes in a double-quoted string:
        \”Rock’n roll\” ”}
```

The output of the example above will be:

```
The following text is double quoted:
        ”Rock and roll!”
The following text is single quoted:
        ‘Rock and roll!’
Using both single and double quotes:
        ”Rock’n roll!”
Using single quotes in a single-quoted string:
        ‘Rock’n roll!’
Using double quotes in a double-quoted string:
        ”Rock’n roll!”
```

Because of the way template code is defined (enclosed in a matching pair of curly braces), the right curly brace (”}”) must be escaped by the backslash character. This must be done regardless of the type of quotes that are used. Note that the left curly brace (”{”) cannot be escaped. The following example demonstrates this.

```
{‘{ This text is inside curly braces.\}’}
```

The output of the template code above will be:

```
{This text is inside curly braces.}
```

Template strings do not support inline expansion of variables (as Perl and PHP do). In other words, it is not possible to mix variables into strings. However, the `concat` operator can be used to append the contents of some variable to a string. This means that this operator can be used to build strings consisting of other strings and / or miscellaneous variables.

Booleans

Booleans are binary; they are either TRUE (1) or FALSE (0). A boolean must be created using either the true or false template operator. For example:

```
{true()}
{false()}
```

In some cases it is possible to use integers as booleans. However, these will not be treated as ”real” booleans. Zero means `FALSE`; all non-zero values mean `TRUE`. Some parts of the system are able to treat an array as if it were a boolean value. While an empty array means `FALSE`, a non-empty array means `TRUE`.

Arrays

Arrays are containers capable of holding a collection of any other variable type, including other arrays. An array can be a simple ordered list or a hash map (that is, an associative array). An element of an ordered list can be accessed using an index number. The number denotes the position of the element inside the array (the first element is zero, the second element is one, and so on). An element of an associative array can be accessed using an index number or an identifier. Regular arrays can be created using the `array` operator. Associative arrays can be created using the hash operator. The following examples demonstrate the creation of arrays and hashes.

Example 1: Array of numbers

```
{array( 2, 4, 8, 16 )}
```

This example creates an array containing four numbers. The array will consist of the following elements:

Index	Value of element
0	2
1	4
2	8
3	16

Example 2: Array of strings

```
{array( 'This', 'is', 'a', 'test' )}
```

This example creates an array containing four strings. The array will consist of the following elements:

Index	Value of element
0	'This'
1	'is'
2	'a'
3	'test'

Example 3: Associative array

```
{hash( 'Red', 16, 'Green', 24, 'Blue', 32 )}
```

This example creates an associative array containing three key-value pairs. The array will consist of the following elements:

Index	Key	Value
0	Red	16
1	Green	24
2	Blue	32

Objects

Template objects are created by PHP code or by special template operators. The system uses objects to represent complex data structures. For example, objects are used to represent information about content nodes, attributes, translations, roles, policies, etc. An object consists of attributes. Each attribute may represent any type of data (number, string, etc.), including another object. The contents of an attribute may be accessed by using the attribute's identifier. The following illustration shows the structure (with example values) of an object (`"ezdate"`) containing information about a date.

Attribute	Type	Value
timestamp	string	567990000
is_valid	boolean	TRUE
year	string	1988
month	string	01
day	string	01

Figure 3.6. The structure of the "ezdate" object

Variable usage

Template variables must be referenced using dollar ($) notation, for example `$my_variable`, `$object_array`, etc. eZ publish template variables are case sensitive. In other words, `$lollipop` is not the same variable as `$LolliPop`. Template variables can be created by the system (from PHP code) or by the author of the template (from within template code). Regardless of how a variable is created, it can be changed using the set function. Some templates have preset variables; for example, the main template (pagelayout) provides access to a collection of variables.

Creating and destroying variables

All variables used in a template must be declared and defined by the `def` function (short for define) before they can be used. A variable exists until the `undef` function (short for "undefine") is used to destroy it. A previously declared variable will be automatically destroyed at the end of the template file in which it was created. The following example demonstrates the most basic use of the `def` and `undef` functions.

```
{def $temperature=32}
    {$temperature}
{undef}
```

The output of the example will be 32. After the `undef` function is called, the `$temperature` variable will not be available. The memory that was allocated to represent the variable will be freed. Both the `def` and the `undef` function can be used with multiple variables at the

same time. In addition, the `undef` function can be used without any parameters. When called without parameters the `undef` function automatically destroys all variables previously created within the template. The following example demonstrates how the `def` and `undef` functions can be used to create and destroy multiple variables at the same time.

```
{def $weather='warm' $celsius=32 $fahrenheit=90}
The weather is {$weather}: {$celsius} C /
        {$fahrenheit} F <br />
{undef $celsius $fahrenheit}
The weather is still {$weather}. <br />
{undef}
```

The output of this example will be:

```
The weather is warm: 32 C / 90 F
The weather is still warm.
```

In the example above, the def function is used to create three new variables: `$temperature`, `$celsius` and `$fahrenheit`. The `undef` function is used twice. The first time, it is used to destroy the `$celsius` and `$fahrenheit` variables. The second time it is called without parameters; thus, the remaining variables (in this case only `$temperature`) will be destroyed. For more information, refer to the online documentation of the `def` and `undef` functions.

Changing the contents of variables

The value of a variable can be changed at any time using the `set` function. It can be used to change the value of any variable, whether it was created by the system or inside a template. No warning will be given if a system variable is changed. The `set` function can be used to change the value of any variable regardless of the variable's current type and the type of the new value. (In other words, this function is capable of changing the type of a variable.)

The `set` function cannot be used to change the value of an element or attribute of an array, hash or object. In fact, the elements and attributes of arrays, hashes and objects cannot be changed from within template code. The following example demonstrates use of the `set` function.

```
{def $weather='warm'}
The weather is {$weather}. <br />
{set $weather='cold'}
The weather is {$weather}.
{undef}
```

The output of the example will be:

```
The weather is warm.
The weather is cold.
```

Like the `def` and `undef` functions, the `set` function can work with multiple variables at the same time. For more information, refer to the online documentation page for the `set` function.

Accessing array elements

The elements of a simple array can only be accessed using numerical indexes. This method is called index lookup. The elements of an associative array can be accessed using the key identifiers. This method is called identifier lookup. In addition, the elements of an associative array can be accessed using index values. The following example demonstrates the different lookup methods.

Index lookup

Index lookup is performed by appending a period (.) and an index number to the name of a simple or associative array. Index lookup may also be performed by appending a pair of brackets that enclose the desired index value. The following example demonstrates how to access the elements of a simple array using index lookup. Note the different syntaxes being used (periods and brackets).

```
{def $sentence=array( 'Life', 'is',
        'very', 'good!' )}
The 1st element is: {$sentence.0} <br />
The 2nd element is: {$sentence.1} <br />
The 3rd element is: {$sentence[2]} <br />
The 4th element is: {$sentence[3]} <br />

{undef}
```

The code above will output the following:

```
The 1st element is: Life
The 2nd element is: is
The 3rd element is: very
The 4th element is: good!
```

Identifier lookup

Identifier lookup can be performed by appending a period and an identifier name to the name of an associative array. Identifier lookup may also be performed by appending a pair of brackets that enclose the desired index value. The following example demonstrates how to access the elements of an associative array using the identifier lookup method. Note the different syntaxes being used (periods and brackets).

```
{def $sentence=hash( 'first', 'Life',
                     'second', 'is',
                     'third', 'very',
                     'fourth', 'good!' )}

The 1st element is: {$sentence.first} <br />
The 2nd element is: {$sentence.second} <br />
The 3rd element is: {$sentence[third]} <br />
The 4th element is: {$sentence[fourth]} <br />

{undef}
```

The following output will be produced:

```
The 1st element is: Life
The 2nd element is: is
The 3rd element is: very
The 4th element is: good!
```

Index lookup and associative arrays

As mentioned above, the elements of an associative array may also be accessed using the index method. The following example demonstrates this.

```
{def $sentence=hash( 'first', 'Life',
                     'second', 'is',
                     'third', 'very',
                     'fourth', 'good!' )}

The 4th element is: {$sentence.3} <br />
The 3rd element is: {$sentence.2} <br />
The 2nd element is: {$sentence[1]} <br />
The 1st element is: {$sentence[0]} <br />

{undef}
```

The following output will be produced:

```
The 4th element is: good!
The 3rd element is: very
The 2nd element is: is
The 1st element is: Life
```

Accessing object attributes

The attributes of an object can only be accessed using the attributes' identifiers. An identifier is the name of an attribute (similar to the keys of an associative array). The following example demonstrates how the different attributes of a node object can be accessed from within a template.

```
The ID of the node: {$node.node_id} <br />

The ID of the object encapsulated by the node:
        {$node.object.id} <br />

The name of the object: {$node.object.name} <br />

First time the object was published:
        {$node.object.published|l10n(
            shortdate )} <br />
```

If the `$node` variable contains a node that has ID number 44 and encapsulates object number 13 named "Birthday" published on the first of April in 2003, the following output will be produced:

```
The ID of the node: 44
The ID of the object encapsulated by the node: 13
The name of the object: Birthday
First time the object was published: 01/04/2003
```

Array and object inspection

By using the attribute template operator, it is possible to quickly inspect the contents of arrays and template objects. The operator creates an overview of available keys, attribute names and / or methods in an object or an array. By default, only the array keys and object attribute names (also called identifiers) are shown. By passing *show* as the first parameter, the operator will also display the values. The second parameter can be used to control the number of levels / children to be returned (the default setting is 2). The following example demonstrates how the operator can be used to inspect the contents of an ”ezcontentobjecttreenode” object.

```
{$node|attribute( show, 1 )}
```

The following output will be produced:

```
Attribute                    Type      Value
-----------------------------------------
node_id                      string        2
parent_node_id               string        1
main_node_id                 string        2
contentobject_id             string        1
contentobject_version        string       10
contentobject_is_published string          1
depth                        string        1
sort_field                   string        8
sort_order                   string        1
priority                     string        5
...
```

As the output shows, a lot of information can be extracted from a node object. In addition to strings and numbers the object also consists of other objects. For example, the creator of the node is an `"ezcontentobject"` object. The creator object can be further inspected by doing the following:

```
{$node.creator|attribute( show, 1 )}
```

The following output will be produced:

```
Attribute                    Type      Value
-----------------------------------------
id                           string       14
section_id                   string        2
owner_id                     string       19
contentclass_id              string        4
is_published                 string        0
...
```

Again, this object contains a lot of information. As mentioned above, the `attribute` operator can be used on both objects and arrays. The following example demonstrates how to inspect the `"data_map"` array (which reveals the object's attributes) of the node's creator object.

```
{$node.creator.data_map|attribute( show, 1 )}
```

The following output (where `"ezcontattrib"` is shown as the short form of `"ezcontentobjectattribute"`) will be produced:

```
Attribute    Type                    Value
-----------------------------------------
first_name    object[ezcontattrib]   Object
last_name     object[ezcontattrib]   Object
user_account object[ezcontattrib]   Object
signature     object[ezcontattrib]   Object
image         object[ezcontattrib]   Object
```

Control structures

The eZ publish template language provides a selection of mechanisms that can be used to solve common programmatic issues such as conditional control, looping, etc. The following list shows an overview of the available mechanisms:

- **IF-THEN-ELSE**
- **SWITCH**
- **WHILE**
- **DO...WHILE**
- **FOR**
- **FOREACH**

IF-THEN-ELSE

The **IF** construct allows for conditional execution of code fragments. It is one of the most important features of many programming languages. The eZ publish implementation makes it possible to do conditional branching using the following elements: **IF**, **ELSE** and **ELSEIF**. The **ELSE** and **ELSEIF** elements are optional. The following examples demonstrate the use of this construct.

Example 1

```
{if eq( $var, 128 )}
    Hello world <br />
{else}
    No world here, move along. <br />
{/if}
```

Example 2

```
{if eq( $fruit, 'apples' )}
    Apples <br />
{elseif eq( $fruit, 'oranges' )}
    Oranges <br />
{else}
    Bananas <br />
{/if}
```

SWITCH

The **SWITCH** mechanism is similar to a series of **IF** statements used within the same expression. This construct is typically useful when a particular variable needs to be compared to different values. It executes a piece of code depending on which value matched a given criteria. The following example demonstrates basic use of this construct.

```
{switch match=$fruits}

    {case match='apples'}
        Apples <br />
    {/case}

    {case match='oranges'}
        Oranges <br />
    {/case}

    {case}
        Unidentified fruit! <br />
    {/case}

{/switch}
```

If the value of the `$fruits` variable is "oranges", the following output will be produced:

```
Oranges
```

WHILE

The **WHILE** construct is the simplest loop mechanism provided by the template language. It tells eZ publish to execute the statements contained inside the **WHILE** command repeatedly, as long as a given expression evaluates to `TRUE`. The value of the expression is checked at the beginning of every loop iteration. If the given expression evaluates to `FALSE`, the statement(s) will not be executed. The following example demonstrates basic use of this construct.

```
{while ne( $counter, 8 )}

    Print this line eight times ({$counter}) <br />
    {set $counter=inc( $counter )}

{/while}
```

If the initial value of $counter is zero, the following output will be produced:

```
Print this line eight times (0)
Print this line eight times (1)
Print this line eight times (2)
Print this line eight times (3)
Print this line eight times (4)
Print this line eight times (5)
Print this line eight times (6)
Print this line eight times (7)
```

DO...WHILE

A **DO...WHILE** loop is similar to a **WHILE** loop, except that the expression is checked at the end of each iteration instead of at the beginning. The main difference is that this construct will always execute the first iteration, regardless of how the test expression evaluates. The following example demonstrates basic use of this construct.

```
{do}
    Keep printing this line ({$counter}) <br />
    {set $counter=inc( $counter )}
{/do while ne( $counter, 8 )}
```

If the initial value of `$counter` is 0, the following output will be produced:

```
Keep printing this line (0)
Keep printing this line (1)
Keep printing this line (2)
Keep printing this line (3)
Keep printing this line (4)
Keep printing this line (5)
Keep printing this line (6)
Keep printing this line (7)
Keep printing this line (8)
```

FOR

Generic looping may be achieved by using **FOR** loops. This construct supports looping over numerical ranges in both directions. It also supports breaking, continual and skipping. The following example demonstrates basic use of this construct.

```
{for 0 to 7 as $counter}
    Value of counter: {$counter} <br />
{/for}
```

The following output will be produced:

```
Value of counter: 0
Value of counter: 1
Value of counter: 2
Value of counter: 3
Value of counter: 4
Value of counter: 5
Value of counter: 6
Value of counter: 7
```

FOREACH

The **FOREACH** construct can be used to iterate over arrays in different ways. The behavior of the command can be changed by using various techniques. The following example demonstrates basic use of this construct.

```
{foreach $objects as $object}
    {$object.name} <br />
{/foreach}
```

The example above will print out the names of the objects stored in the `$objects` array. If this array stores four objects with the names `"Emmett Brown"`, `"Marty McFly"`, `"Lorraine Baines"` and `"Biff Tannen"`, the following output will be produced:

```
Emmett Brown
Marty McFly
Lorraine Baines
Biff Tannen
```

Functions and operators

The eZ publish template language provides a collection of various functions and operators that can be used to perform various tasks. In addition, it is possible to extend the system by creating custom operators for special needs. Custom operators are programmed in PHP; an example is provided in the *Extensions* chapter.

Template functions

A function takes a set of named parameters, performs a specific task and returns a result. It can be called anywhere in a template using the following syntax:

```
{function_name parameter1=value1
        parameter2=value2 ...}
```

A function takes zero or more named parameters. The parameters must be specified after the function name, separated by spaces. Since each parameter is specified using the parameter's name, the parameters can be provided in any order. Each parameter must be assigned a value using the equals sign. The following illustration shows a typical usage of a commonly used function.

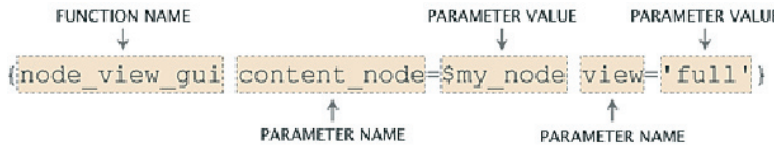


Figure 3.7. Typical components of a function call

The example above calls the `node_view_gui` function. This function displays a node by including the template associated with the view mode. The node is specified using the *`content_node`* parameter. The desired view mode is specified using the *`view`* parameter.

Template operators

An operator takes unnamed parameters, performs a specific task and returns a result. An operator is capable of handling a parameter that is passed to it using a pipe. It can be called anywhere in a template using the following syntax:

```
{$input_parameter|operator_name( parameter1,
        parameter2 ... )}
```

Because the operator only uses unnamed parameters, the parameters must be specified in the order dictated by the operator's documentation page. In addition, the parameters must be separated by commas. The following illustration shows a typical usage of a commonly used operator.

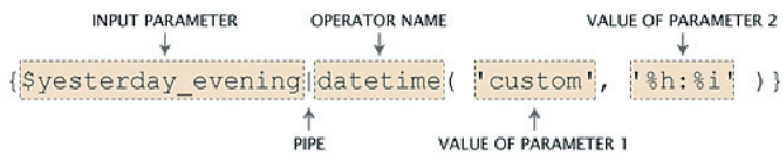


Figure 3.8. Typical components of a template operator call

The example above demonstrates the usage of the `datetime` operator. This operator can be used to convert a UNIX timestamp to a human-readable format. The timestamp is provided by the `$yesterday_evening` variable as the input parameter. The first parameter tells the operator that the output should be formatted using a custom schema. The schema is defined by the second parameter (hours : minutes).

Piping

An operator takes input on the left side and produces output on the right side. A collection of operators can be glued together using pipes. A pipe ensures that the output from one operator is presented as the input parameter to another operator. The following example demonstrates how pipes and operators can be used to create a string.

```
{concat( 'To ', 'The ' )|prepend( 'Back ' )
        |append( 'Future' )}
```

The following output will be produced:

```
Back To The Future
```

Basic template tasks

This section discusses some common issues related to template development.

Template inclusion

A template file can be included using the `include` function. Since this function makes it possible to include any file from any location within the eZ publish directory, the function must be told that it should look for the file within the *`design/`* directory. This can be done by prefixing the path or filename with `"design:"`. The following example demonstrates how the include function can be used to include a template file called `footer.tpl` that is located in the *`templates/`* directory of a design.

```
{include uri='design:footer.tpl'}
```

If the requested file is not found within the main design of the siteaccess, the system will search for it in the additional designs and the standard design. Refer to the *Automatic fallback* section of the *Concepts and basics* chapter for more information about this feature.

Output washing

Variables that may contain invalid strings should always be cleared using the `wash` operator. This operator ensures that the output does not contain any elements that should not be included in the HTML generated by eZ publish. The following example demonstrates how the `wash` operator works.

```
{def $bogus_string='hello < world'}
{$bogus_string|wash()}
```

The following HTML output will be produced:

```
hello &lt; world
```

E-mail address obfuscation

In addition to ensuring proper output, the wash operator can be used to obfuscate e-mail addresses on a web page. An obfuscated e-mail address has less chance of being harvested by a robot searching for e-mail addresses for spammers. The following example demonstrates how the `wash` operator can be used with an e-mail address.

```
{def $email_address='allman@example.com'}
{$email_address|wash( 'email' )}
```

The following output will be produced:

```
allman[at]example[dot]com
```

String concatenation

The `concat` operator makes it possible to glue several strings together to produce a single string. The following example demonstrates how this operator works.

```
{def $my_string='sausage'}
{concat( 'Liver ', $my_string, ' sandwich' )}
```

The following output will be produced:

```
Liver sausage sandwich
```

Custom view parameters

The URL of a node view request may contain custom parameters. The custom view parameters must be specified at the end of the URL using a special notation. For each parameter, a name and value must be specified. The name must be encapsulated by parentheses. Each element must be separated by slashes. The following example demonstrates how custom parameters can be used (in addition to the view parameters) in a system URL that requests a node.

```
http://www.example.com/content/view/
        full/13/(color)/green/(amount)/34
```

The same parameters can be appended to the virtual URL of the node:

```
http://www.example.com/company/
        about_us/(color)/green/(amount)/34
```

When custom view parameters are used, the system will create an associative array using the names of the provided parameters as the keys. All parameter values will be treated as strings. The array will be represented by the `$view_parameters` variable in the template. The parameters given in the examples above would produce an associative array with the following contents:

Key	Type	Value
`color`	string	green
`amount`	string	34

The following example demonstrates how the custom view parameters can be accessed in the template used to display the node.

```
The color is: {$view_parameters.color} <br />
The amount is: {$view_parameters.amount} <br />
```

The following output will be produced:

```
The color is: green
The amount is: 34
```

URL handling

Whenever a link, a non-content specific image, a stylesheet, etc., is to be included, a suitable template operator must be used to ensure that the path to the included file is correct. At any time, one of the following operators should be used:

- `ezurl`

- ezimage
- ezdesign

ezurl

The ezurl operator ensures that a URL works regardless of the location of the eZ publish folder, the access method and the environment in which eZ publish is running (non-virtual host, virtual host, etc.). Only the eZ publishspecific part of the URL needs to be provided. The rest (http://, host, domain, directory, siteaccess, port, etc.) will be generated by the operator. The final output will be a valid address. This approach makes it possible to use generic URLs in templates without the risk of having to modify every address when the site is moved and / or when the access method is changed. By default, the ezurl operator outputs an address that is already enclosed by two double quotes. In other words, the output can be fed directly to a hyperlink reference in the HTML code. The following examples demonstrate the usage of this operator.

Link to a module / view (using a system URL)

```
<a href={'/user/login'|ezurl()}>Login</a>
```

The example above demonstrates how to create a link to the login view of the user module. The "user/login" is one example; another example is a link to a node, such as "content/view/full/34". If eZ publish is running in a directory called *ezpublish* on www.example.com using the URL access method, and the name of the siteaccess is *"my_company"*, the operator will produce the following output:

```
http://www.example.com/ezpublish/index.php/
        my_company/user/login
```

If eZ publish is running in virtual host mode and uses the host access method, the following URL will be produced:

```
http://www.example.com/user/login
```

Link to a node (using the node's virtual URL)

When a link to a node (using the node's virtual URL, also known as "URL alias") is created, the address must be piped through the ezurl operator. The reason for this is that the internal URL table only contains the eZ publish-specific part of the URLs. The following example demonstrates how to use the ezurl operator to create a valid virtual URL for a node.

```
<a href={$node.url_alias
        |ezurl()}>Link to a node</a>
```

If the URL alias of the node is company/about_us and eZ publish is running in a virtual host environment using the host access method, the following URL will be produced:

```
http://www.example.com/company/about_us
```

For information about how eZ publish treats URLs, refer to the *URL translation* section of the *Concepts and basics* chapter.

ezimage

The `ezimage` operator works in the same way as the `ezurl` operator (described above), except that it does not include the *`index.php`* part. This operator must be used every time a non-content specific image is included in a template. The image must be placed in the *`images/`* directory of one of the designs used by the siteaccess. The operator produces a valid link to the image regardless of the directory, access method and / or the environment in which eZ publish is running. The following example demonstrates how `the ezimage` operator should be used.

```
<img src={'women.jpg'|ezimage()}
        alt="This is my image." ... />
```

If eZ publish is using the host access method and the siteaccess is using a design called `"my_design"`, the operator will produce the following output:

```
http://www.example.com/design/my_design/
        images/women.jpg
```

If the image is placed inside a subdirectory within the *`images/`* directory, the name of the subdirectory must be specified in the template. If the requested file is not found within the main design of the siteaccess, the system will search for it in the additional designs and the standard design. Refer to the *Automatic fallback* section of the *Concepts and basics* chapter for information about this feature.

ezdesign

The `ezdesign` operator works in the same way as the `ezurl` operator (described above), except that it does not include the *`index.php`* part. This operator must be used every time a design element (style sheets, JavaScript, etc.) is included in a template. The operator produces a valid link for the given design component by providing the root to the design directory which contains the target file. The following example demonstrates the proper way of including a CSS file using this operator.

```
...
<style type="text/css">
    @import url({'stylesheets/my_stuff.css'
        |ezdesign});
</style>
...
```

If eZ publish is using the host access method and the siteaccess is using a design called `"my_design"`, the operator will produce the following output:

```
http://www.example.com/design/
        my_design/stylesheets/my_stuff.css
```

If the requested file is not found within the main design of the siteaccess, the system will search for it in the additional designs and the standard design. Refer to the *Automatic fallback* section of the *Concepts and basics* chapter for more information about this feature.

Information extraction

Information stored by eZ publish can be extracted using the `fetch` template operator. This operator gives access to the fetch functions provided by a module. It is typically used to extract nodes, objects, etc. using the content module. The fetch operator can only be used with modules that provide support for data fetching. Refer to the *Fetch functions* section of the online documentation for a complete overview of available fetch functions. The following model and table shows the usage and parameters of the fetch operator.

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

Parameter	Description
module	The name of the target module.
function	The name of the fetch function within the target module.
parameters	An associative array containing the function parameters.

A module's `fetch` functions and parameters are defined in the `function_definition.php` file within the directory of the module.

Fetching a single node

The following example demonstrates how the fetch operator can be used to extract a single node from the database.

```
{def $my_node=fetch( content, node,
        hash( node_id, 13 ) )}
...
{undef}
```

The example above instructs eZ publish to fetch a single node from the content module. Only one parameter is given, which is the ID number of the node to be fetched. The operator will return an `ezcontentobjecttreenode` object which will be stored in the `$my_node` variable. This variable can then be used to extract information about the node and the object that it encapsulates. For example, it is possible to extract the name, attributes and the time when the object was published. If the node is unavailable or non-existing, or if the currently logged-in user doesn't have read access to it, the operator will return a `FALSE` boolean value. Refer to the *Objects* section of the online reference documentation for an overview of what the different objects provide.

Fetching multiple nodes

It is possible to fetch all the nodes that are directly below a specific node. This can be done by using *`list`* instead of *`node`* as the second parameter to the `fetch` operator. The following example demonstrates how the fetch operator can be used to extract all the nodes directly below node number 13.

```
{def $my_node=fetch( content, list,
        hash( parent_node_id, 13 ) )}
...
{undef}
```

The operator will return an array of `ezcontentobjecttreenode` objects. The `list` fetch function of the `content` module can take several parameters. These parameters are optional and can be used to fine-tune the fetch, for example by filtering out specific nodes. The following table gives an overview of the most commonly used parameters.

Parameter	Description
sort_by	The method and direction that should be used hen the nodes are sorted (must be specified as an array).
limit	The number of nodes that should be fetched.
offset	The offset at which the fetch should start.
class_filter_type	The type of filter that should be used, either `"include"` or `"exclude"`.
class_filter_array	The types of nodes that should be included or excluded by the filter (must be specified as an array).

The following example demonstrates how to fetch an alphabetically sorted array of the ten latest articles directly below node number 13.

```
{def $my_node=fetch( content,
        list,
        hash( parent_node_id, 13,
              limit, 10,
              class_filter_type, include,
              class_filter_array, array
                     ( 'article' ) ) )}
...
{undef}
```

Refer to the online documentation for the `list` fetch function for a complete description of the available parameters and examples of usage.

Outputting node and object data

Once an `ezcontentobjecttreenode` object representing a node is available in a template variable, it can be used to output information about the node and the contents of the object that the node encapsulates. The following section demonstrates the extraction of the most common elements.

General information

The name of the object

```
{$node.name|wash}
```

The name of the object is directly available through the node (in other words, it is possible to reach it by `$node.name` instead of `$node.object.name`). The `wash` operator is used for ensuring that the output doesn't contain any invalid characters and / or sequences that may invalidate the HTML.

The date / time when the object was first published

```
{$node.object.published|l10n( 'shortdatetime' )}
```

Since the publishing value is stored as a UNIX timestamp, it must be properly formatted for output. This can be done by using the `l10n` operator, which makes it possible to format different types of values according to the current locale settings.

The date / time when the object was last modified

```
{$node.object.modified|l10n( 'shortdatetime' )}
```

Since the modification value is stored as a UNIX timestamp, it must be properly formatted for output. This can be done using the `l10n` operator, which makes it possible to format different types of values according to the current locale settings.

The name of the user who initially created the object

```
{$node.object.owner.name|wash}
```

The name of the user who last modified the object

```
{$node.object.current.creator.name|wash()}
```

The name of the class of which the object is an instance

```
{$node.object.class_name|wash()}
```

Object attributes

The attributes of an object can be extracted by using the `data_map` method. This method returns an associative array of `ezcontentobjectattribute` objects, where each object represents one attribute. The keys of the array are the class attribute identifiers. The following example demonstrates how an attribute called `first_name` can be accessed using the object's data map.

```
{$node.object.data_map.first_name}
```

The example above will not produce any valuable output because the requested data needs to be formatted. There are two ways of outputting the contents of attributes:

- Raw output (the `.output` extension)
- Formatted output (the `attribute_view_gui` function)

The main difference between raw and formatted output is that formatted output makes use of a template which in turn outputs the requested data. Raw output simply outputs the data within the same template from which the request for output was issued. Output should always be presented through the `attribute_view_gui` function. The raw output method should only be used when / if necessary (for example when checking the value of an attribute using an **IF** statement).

Raw output

Raw output is a raw dump of the contents that are stored by the attribute. The syntax depends on the datatype that represents the attribute. In most cases, it is possible to generate the output by appending `.output` to the identifier.

Generic solution

The following example demonstrates how to output the contents of an attribute called `my_attribute`.

```
{$node.object.data_map.my_attribute.content}
```

XML block

The following example demonstrates how to output the contents of an XML block called `my_xml`.

```
{$node.object.data_map.my_xml.
        content.output.output_text}
```

Image

The following example demonstrates how to output an image stored by an attribute called `my_image`.

```
<img src="{$node.object.data_map.my_image.content
        [image_size].full_path}" ... />
```

Formatted output

Each datatype has a set of templates that are used to display the contents in different contexts. There are at least two templates for each datatype: a view template and an edit template. While the view template is used to display information, the edit template is used when the data is being edited. The default templates for the datatypes are located within the standard design: *`design/standard/templates/content/datatype/`*.

The `attribute_view_gui` function makes it possible to display the contents of an attribute by inserting the view template of the datatype used by the attribute. The following example demonstrates how this function can be used.

```
{attribute_view_gui attribute=
        $node.object.data_map.name_of_any_attribute}
```

The example above will generate proper output for any attribute regardless of the datatype.

The template override system

The template override system makes it possible to use templates other than the default ones (specified in the code for the different views and templates). This mechanism allows the creation of template overrides for virtually any template used by eZ publish, including templates that are requested by the `include` template function using the `"design:"` prefix. In particular, template overrides are useful for displaying different types of nodes in different ways.

An override for a view template is usually activated by a set of conditions. If the conditions match, the alternate template will be used. Different views provide different conditions; some views do not provide any conditions at all. Refer to the *Template override conditions* section of the online documentation for a description of available match rules. The most flexible set of conditions are provided by the `view` view of the `content` module (used when a node is displayed). The following illustration shows how the override mechanism plugs into the rest of the system.

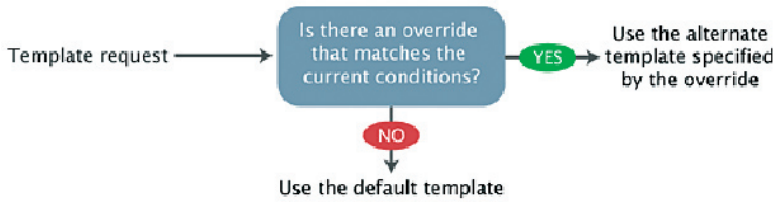


Figure 3.9. The override system

The template overrides must be defined in the *override.ini.append.php* file of a siteaccess. This file consists of override blocks. A block is a named set of rules that tells eZ publish to use an alternate template in a specific situation. For each block, the following information must be specified:

- A unique name for the override
- The template that should be overridden
- The template that should be used instead of the one being overridden
- The name of the directory in which the override template resides (usually *templates/*)
- A set of conditions / rules that control when the override should be activated.

Note that the rules / conditions are optional. If no rules are specified, the override will always be active. The following illustration shows a typical example of a template override with additional explanations.

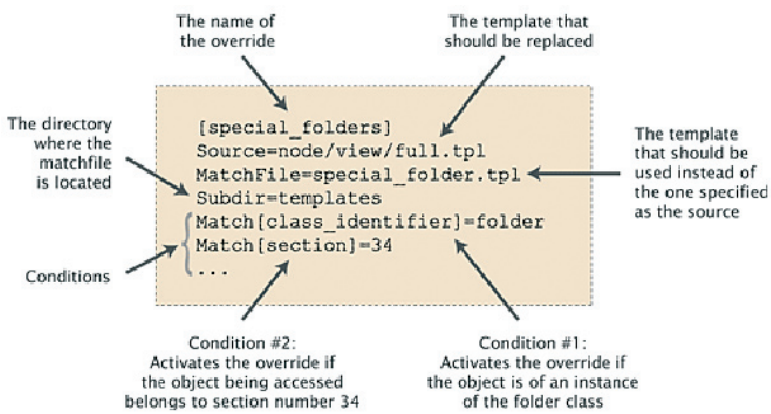


Figure 3.10. Template override example

The example above defines an override called `special_folders`. This override will be used when the system receives a request to display a node using full view. The override will only be activated if the object referenced by the node is an instance of the folder class and if it belongs to section number 34. When the override is activated, the system will attempt to use the alternate template (*override/templates/special_folder.tpl*, located in the main design). If eZ publish is unable to find the alternate template, it will look for it in the additional designs and the standard design. Refer to the *Automatic fallback* section of the *Concepts and basics* chapter for more information about this feature.

Multiple / conflicting overrides

The priorities of the overrides are determined by their positions in the file. If there are several overrides with similar or equal rules, eZ publish will use the first override that matches and thus the rest of the overrides will be omitted. Because of this, overrides that are, for example, activated on a node ID or an object ID basis should always be placed first. Otherwise they might never be triggered because of the presence of a more generic override with a higher priority.

Template override example

The following example demonstrates how the template override system can be used to display alternate templates in different situations.

Imagine a simple content tree consisting of two folders: ”News” and ”Products”. The ”News” folder contains news articles and the ”Products” folder contains products. The following illustration shows an example of such a tree.

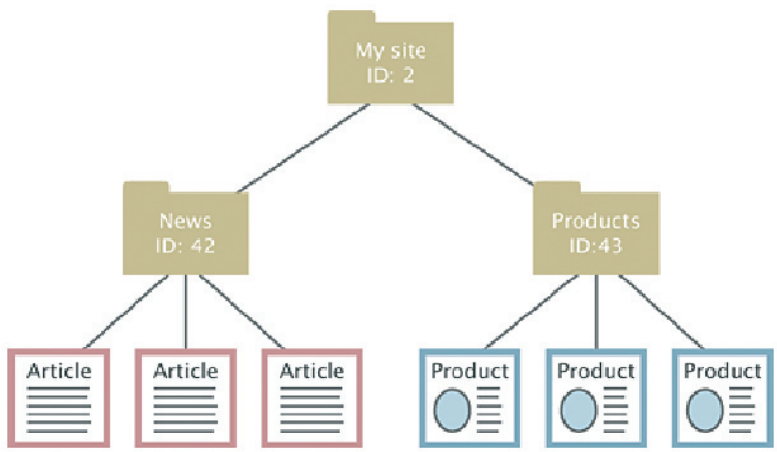


Figure 3.11. Example content node tree

Without any overrides, eZ publish will most likely display all nodes using the same template. This would probably be the default full view template located in the standard design. However, what if we wish to display custom / alternate templates for the different nodes? We would perhaps like the system to behave in the following way:

- Display a special ”welcome” template when the ”My site” node is accessed.
- Display a custom folder template when a folder is accessed.
- Display a custom article template when a news article is accessed.
- Display a custom product template when a product is accessed.

The requests in the list above can be easily fulfilled by creating a couple of overrides. The welcome page should be approached by using an override that is triggered by the identification number of the ”My site” node. The rest of the requests can be approached by using the class identifier key, which allows an override to be triggered when an object of a certain class is accessed. The following example shows the contents of an `override. ini.append.php` file that makes this possible:

```
# Override for welcome page
[welcome_page]
Source=node/view/full.tpl
MatchFile=welcome.tpl
Subdir=templates
Match[node]=2
# Override for folders
[my_folder]
Source=node/view/full.tpl
MatchFile=my_folder.tpl
Subdir=templates
Match[class_identifier]=folder
# Override for articles
[news_articles]
Source=node/view/full.tpl
MatchFile=my_article.tpl
Subdir=templates
Match[class_identifier]=article
# Override for products
[products]
Source=node/view/full.tpl
MatchFile=my_product.tpl
Subdir=templates
Match[class_identifier]=product
```

The alternate templates should be stored in the *override/templates/* subdirectory of the main design used by the siteaccess. The following illustration shows where the templates would be located in a design called ”example”.

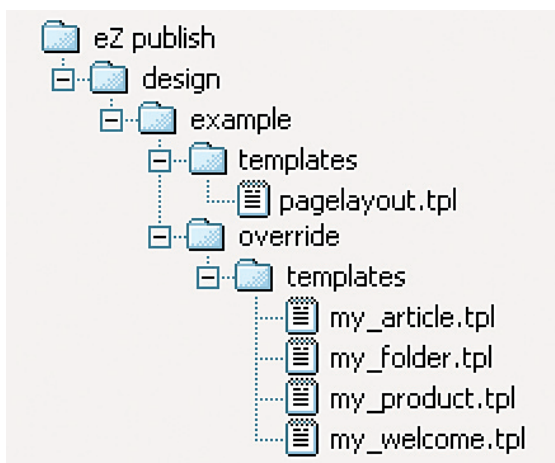


Figure 3.12. Pagelayout + override templates in example design

When the system is in use, the different overrides would be activated based on the given conditions. The following illustration shows where and when the different alternate templates would be used.

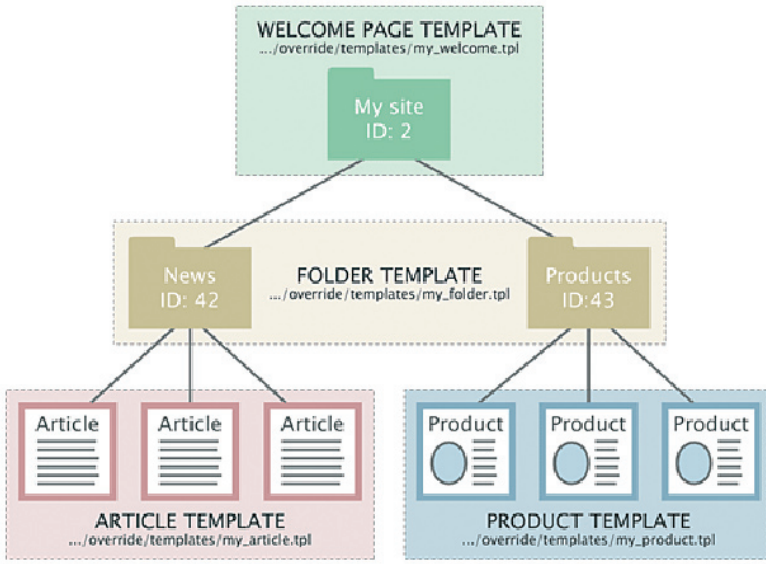
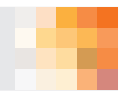


Figure 3.13. Template override example

Every time a node referencing a folder object is viewed, the system will use the *my_folder.tpl* template. When an article is viewed, the `my_article.tpl` template will be used. When a product is viewed, the `my_product.tpl` template will be used. When node number 2 (the ”My site” node) is viewed, the `my_welcome.tpl` will be used.

Chapter 4. Common solutions



This chapter contains a collection of common solutions, tips and tricks which will be useful for a rookie eZ publish developer. Without delving into too much detail, this chapter contains the information a typical developer will need to create a basic but feature-rich eZ publish site. Among other things, the following topics are covered:

- How to configure siteaccesses for new sites
- How to include breadcrumbs in the pagelayout
- How to add a search interface to the pagelayout
- How to configure the search results
- How to reindex the search
- How to include and use the page navigator
- How to reset the administrator password
- How to change the username of a user
- How to add login functionality to a site
- How to create a protected area
- How to create and use custom XML tags
- How to back up and restore a site

Setting up siteaccesses for a new site

A typical eZ publish site consists of two *siteaccesses*: a public interface for visitors and a restricted interface for administrators. The following steps describes the process for creating a new site. Refer to the *installation* chapter for further details regarding commands, paths, etc.

1. Create a new database for the new site.
2. Initialize the database using the *kernel_schema.sql* and the *cleandata.sql* scripts.
3. Create the directories for the siteaccesses:

   ```
   settings/siteaccess/example/
   ```

   ```
   settings/siteaccess/example_admin/
   ```
4. Create the *site.ini.php* configuration override for the ”example” siteaccess (under *settings/siteaccess/example/*):

```
<?php /* #?ini charset="iso-8859-1"?

[SiteSettings]
SiteName=Example
SiteURL=www.example.com
IndexPage=/content/view/full/2
LoginPage=embedded

[DatabaseSettings]
DatabaseImplementation=ezmysql
Server=localhost
User=root
Password=
Database=example

[DesignSettings]
SiteDesign=example

[FileSettings]
VarDir=var/example

[SiteAccessSettings]
RequireUserLogin=false
ShowHiddenNodes=false

[MailSettings]
AdminEmail=webmaster@example.com
EmailSender=webmaster@example.com

[InformationCollectionSettings]
EmailReceiver=webmaster@example.com

[UserSettings]
RegistrationEmail=webmaster@example.com

[TemplateSettings]
TemplateCache=disabled
TemplateCompile=disabled
ShowXHTMLCode=disabled
Debug=disabled

[ContentSettings]
ViewCaching=disabled

[RegionalSettings]
Locale=eng-GB
ContentObjectLocale=eng-GB
TextTranslation=disabled
[DebugSettings]
```

```
DebugOutput=enabled
Debug=inline
DebugRedirection=disabled

*/ ?>
```

These settings instruct eZ publish to do the following when the ”example” siteaccess is in use:

- Use a database called ”example”
- Use a design called ”example”
- Use a var directory called *"example"*
- Allow anonymous visitors browse the site
- Hike nodes marked as hidden
- Send all mail generated by the system to `"webmaster@example.com"`
- Do not use any caching mechanisms
- Do not show the template debug messages
- Use British English as the default language for content
- Use British English when displaying interfaces, sending mail, etc.
- Show debug messages at the bottom of every page
- Do not debug internal redirections

5. Create the configuration overrides for the ”example_admin” siteaccess. The quickest and easiest way is to do this as follows:
   - Copy the contents of the *settings/siteaccess/admin/* directory (the one that was generated by the Setup Wizard) to the *settings/siteaccess/example_admin/* directory
   - Edit *site.ini.append.php* and ensure that it uses the correct database, var directory, e-mail addresses, etc.

   In other words, the *site.ini.append.php* file for the ”example_admin” siteaccess (under *settings/siteaccess/example_admin/*) should look like this:

```
<?php /* #?ini charset="iso-8859-1"?

[SiteSettings]
SiteName=Example
SiteURL=www.example.com
IndexPage=/content/view/full/2
LoginPage=embedded

[DatabaseSettings]
DatabaseImplementation=ezmysql
Server=localhost
User=root
Password=
Database=example
```

```
[DesignSettings]
SiteDesign=admin

[FileSettings]
VarDir=var/example

[SiteAccessSettings]
RequireUserLogin=true
ShowHiddenNodes=true

[MailSettings]
AdminEmail=webmaster@example.com
EmailSender=webmaster@example.com

[InformationCollectionSettings]
EmailReceiver=webmaster@example.com

[UserSettings]
RegistrationEmail=webmaster@example.com

[TemplateSettings]
TemplateCache=enabled
TemplateCompile=enabled
ShowXHTMLCode=disabled
Debug=disabled

[ContentSettings]
ViewCaching=enabled
CachedViewPreferences=[- truncated -]

[RegionalSettings]
Locale=eng-GB
ContentObjectLocale=eng-GB
TextTranslation=disabled

[DebugSettings]
DebugOutput=disabled
Debug=inline
DebugRedirection=disabled

*/ ?>
```

Note that the value of `CachedViewPreferences` should be copied from *`settings/siteaccess/admin/site.ini.append.php`*. These settings instruct eZ publish to do the following when the ”example_admin” siteaccess is in use:

- Use a database called ”example”
- Use the built-in admin design
- Use a var directory called *`"example"`*
- Do not allow anonymous visitors to browse the site
- Show hidden nodes

- Send all mail produced by the system to "webmaster@example.com"
- Use all caching mechanisms
- Do not show template debug messages
- Use British English as the default / primary language for content
- Use British English when displaying interfaces, sending mail, etc.
- Do not show any debug messages
- Do not debug internal redirections

6. Set up the global configuration override for *site.ini (in settings/override/site.ini.append.php*):

```
<?php /* #?ini charset="iso-8859-1"?

[Session]
SessionNameHandler=custom

[SiteSettings]
DefaultAccess=example
SiteList[]
SiteList[]=example

[SiteAccessSettings]
CheckValidity=false
AvailableSiteAccessList[]
AvailableSiteAccessList[]=example
AvailableSiteAccessList[]=example_admin
MatchOrder=uri

[MailSettings]
Transport=sendmail

*/ ?>
```

The global configuration override has higher priority than the siteaccess configurations. These settings instruct eZ publish to do the following at all times (for both siteaccesses in our case):

- Use a custom session name handler
- Default to the "example" siteaccess if unable to determine which siteaccess should be used
- If WebDAV is enabled, allow clients to access the content used by the "example" siteaccess
- Do not run the web-based Setup Wizard
- Allow the usage of the "example" and "example_admin" siteaccesses
- Determine which siteaccess to use based on the first parameter provided after *index.php* in the requested URLs
- Use a local application called sendmail (or a sendmail-compatible solution on UNIX systems) to deliver mail

Windows users should route outgoing mail through an SMTP server. This can be done by making use of the following settings:

```
[MailSettings]
Transport=SMTP
TransportServer=mail.example.com
TransportPort=25
TransportUser=john
TransportPassword=secret
```

7. Clear all caches by running the following PHP script from within the root of the eZ publish directory:

   ```
   ./bin/php/ezcache.php --clear-all
   ```

   Note that this requires the presence of a command line interpreter for PHP scripts. The cache can also be deleted manually by removing directories called *cache/* inside the *var/* directory (and deeper within the structure) or by using a UNIX shellscript called *clearcache.sh* located in *bin/shell/*.

8. Test the solution by accessing both the admin and the public siteaccess. The public siteaccess should fallback to the default / standard design.

Setting up a virtual host-based solution

A virtual host setup together with the *host* access method is the best and most secure way to run eZ publish on an Apache web server. In addition, the virtual host mechanism makes it possible to run several sites on the same server without the need for multiple IP addresses. The following example demonstrates how a system can be set up in order to function properly in a virtual host environment. Consult the virtual hosts section of the Apache documentation for more information.

1. Configure the DNS server so that it resolves the desired domains (for example `"www.example.com"` and `"admin.example.com"`) to the IP address of the web server. Alternatively, if you are just testing the solution, you can edit your local *hosts* file and add the necessary entries there.

2. Configure the web server so that it uses the desired virtual host configuration when the site is being accessed (in this case when either `"www.example.com"` or `"admin.example.com"` is accessed). The following example shows the lines that must be added to the *httpd. conf* file. It assumes that eZ publish is installed under */var/www/example/*, and that the IP address of the web server is ”62.70.12.230”.

   ```
   NameVirtualHost 62.70.12.230
   ```

   ```
   <VirtualHost 62.70.12.230>
     <Directory /var/www/example>
       Options FollowSymLinks
    AllowOverride None
   </Directory>
   ```

```
<IfModule mod_php4.c>
  php_admin_flag safe_mode Off
  php_admin_value register_globals 0
  php_value magic_quotes_gpc 0
  php_value magic_quotes_runtime 0
  php_value allow_call_time_pass_reference 0
</IfModule>

DirectoryIndex index.php

<IfModule mod_rewrite.c>
  RewriteEngine On
  Rewriterule ^/var/storage/.* - [L]
  Rewriterule ^/var/[^/]+/storage/.* - [L]
  RewriteRule ^/var/cache/texttoimage/.* - [L]
  RewriteRule ^/var/[^/]+/cache/texttoimage/.*
        - [L]
  Rewriterule ^/design/[^/]+/(stylesheets|
        images|javascript)/.* - [L]
  Rewriterule ^/share/icons/.* - [L]
  Rewriterule ^/extension/[^/]+/design/[^/]+/
        (stylesheets|images|javascripts?)/.*
        - [L]
  Rewriterule ^/packages/styles/.+/
        (stylesheets|images|javascript)/
        [^/]+/.* - [L]
  RewriteRule .* /index.php
</IfModule>

 DocumentRoot /var/www/example
 ServerName www.example.com
 ServerAlias admin.example.com
</VirtualHost>
```

Note that it isn't necessary to create a separate virtual host block for "`admin.example.com`"; it can be added to the existing block using the `ServerAlias` directive.

3. Make sure that the newly added virtual host configuration is active by restarting the web server or making it reload the configuration files. Before restarting the service, it is a good idea to run **apachectl configtest** which verifies the syntax of the configuration file. This allows you to fix problems that would lead to the web server being unavailable (as the service will not start if there are errors in the configuration file).

4. Configure eZ publish to use the host access method. This can be done using the web-based Setup Wizard during installation, or by editing the global configuration override of *site.ini* (*settings/override/site.ini.append.php*). The following example shows the lines that should be added (remove similar lines if they are present).

```
[SiteAccessSettings]
AvailableSiteAccessList[]
AvailableSiteAccessList[]=example
AvailableSiteAccessList[]=example_admin
MatchOrder=host

HostMatchMapItems[]=www.example.com;example
HostMatchMapItems[]=admin.example.com;
         example_admin
```

This configuration tells eZ publish that it should use the ”example” siteaccess if a request starts with `"www.example.com"` and ”example_admin” if the request starts with `"admin.example.com"`. For more information about site management in eZ publish, refer to the *Site management* section of the *Concepts and basics* chapter.

5. Clear all caches and test the solution. If everything works, `"www.example.com"` should bring up the public / user siteaccess and `"admin.example.com"` should bring up the Administration Interface.

Periodic and scheduled maintenance

Some features of eZ publish depend on a maintenance script that takes care of various tasks behind the scenes. This script is located in the root of the eZ publish directory and should be executed at regular intervals. The script is called *runcronjobs.php*. Among other things, it processes workflows, checks / validates URLs, sends out notification e-mails, etc. Although eZ publish works without a periodical execution of *runcronjobs.php*, it is still recommended to have it running in the background. Some features, for example the notification system, will not be available if the script is not running. The most common practice is to instruct the operating system (or some application) to automatically run the script every 15 minutes. On UNIX/Linux systems this can be done by making use of ”cron”. On Windows, the script can be run by the ”Scheduled Tasks” service.

Cronjobs on UNIX / Linux

Cron is the name of a utility that allows the automatic execution of tasks in the background. It is typically used for periodic system administration and maintenance tasks (for example, creating a weekly backup). A program often referred to as the *cron daemon* is running silently in the background, spending its time waiting and executing cron jobs. A *cron job* is a script or a command that is run at specified intervals by the daemon. The cron jobs must be set up in a crontab. A *crontab* is a text file that contains information about the intervals and the tasks that should be executed. The following example shows how a cronjob for eZ publish should be set up in the crontab. It assumes that eZ publish is located in */var/www/ezpublish/*, that the PHP command line interface program is */usr/local/bin/php* and that the name of the target siteaccess is ”example”.

```
# The path to the eZ publish directory.
EZPUBLISH=/var/www/ezpublish
```

```
# Location of the PHP command line interface binary.
PHPCLI=/usr/local/bin/php

# Instruct cron to run "runcronjobs.php"
# every 15 minutes
0,15,30,45 * * * * cd $EZPUBLISH && $PHPCLI -C
        runcronjobs.php -q -s example 2>&1
```

When added to the crontab, the cron daemon will run the *runcronjobs.php* script using the PHP command line interface binary every 15 minutes. The *-q* parameter instructs the script to run in quiet mode (suppressing unnecessary output). The *-s example* indicates which siteaccess configuration the script should use. The "2>&1" notation instructs the system to combine standard output and error messages into one stream.

Scheduled tasks on Windows

Unlike UNIX/Linux systems, Windows does not provide access to cron. Instead, Windows has its own solution called *Scheduled Tasks*. A scheduled task can be set up by selecting *Scheduled Tasks* from the Control Panel. This will bring up a wizard that asks what should be executed, when, etc. It should be configured to run a batch (*.bat*) file every 15 minutes. The batch file should navigate into the eZ publish directory and run the script as described in the previous example.

Creating a custom design

The following list of steps explains how a custom design can be created.

1. Create a new design directory:

   ```
   design/example/
   ```

2. Create the necessary directories inside the new design directory:
   - *stylesheets/*
   - *images/*
   - *templates/*
   - *override/templates/*

3. Add a custom pagelayout (in *design/example/templates/pagelayout.tpl*):

   ```
   <!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>

   <style type="text/css">
   ```

```
        @import url({'stylesheets/core.css'
                |ezdesign});
        @import url({'stylesheets/debug.css'
                |ezdesign});
    </style>

    {include uri='design:page_head.tpl'}

    </head>

    <body>

    <h1>Hello world</h1>

    {$module_result.content}

    <!--DEBUG_REPORT-->

    </body>

    </html>
```

4. Make sure that the custom design is actually used by providing its name for the `SiteDesign` directive in the configuration override of *`site.ini`* for the siteaccess.
5. Clear all caches and verify that the custom design is used; you should be able to see your own pagelayout containing the "Hello world" message.

Adding images

The following example shows how an image that links back to the root of the site can be added to the pagelayout (or any other template). Note the usage of the ezurl operator.

```
<a href={'/'|ezurl}>
    <img src={'example.png'|ezimage}
            alt="Example." />
</a>
```

Creating a simple menu

The following example shows how a basic menu can be created within the pagelayout. The code extracts folder nodes that are directly below the content root. It then loops through the fetched nodes and displays their names as links. A simple checking mechanism detects whether the node being viewed (the current node) is one of the menu items or not. If it is, it will not be displayed as a link because it is already being viewed.

```
{* Fetch folders below the content root. *}
{def $elements=fetch( content, list,
        hash( parent_node_id, 2,
              class_filter_type, include,
              class_filter_array,
                    array( 'folder' ) )}
```

```
{* Loop through the fetched nodes. *}
{foreach $elements as $element}

    {* Check if a node is being viewed. *}
    {if $module_result.node_id}

        {* Check if the node is a menu item. *}
        {if eq( $module_result.node_id,
              $element.node_id )}

            {* Display name of the node. *}
            {$element.name|wash}

        {else}

            {* Display name of node as link. *}
            <a href={$element.url_alias|ezurl}>
                     {$item.name|wash}</a>

        {/if}

    {else}

            {* Display name of node as link. *}
            <a href={$element.url_alias|ezurl}>
                     {$item.name|wash}</a>

        {/if}

    {/foreach}
```

Adding custom templates

There are two ways of adding custom templates to a design: by making use of the override mechanism, or by adding a modified system template to the custom design. The first method is useful when there is a need to show different kinds of nodes in different ways. The second method is preferable when there is a need to use modified versions of miscellaneous system templates (for example a customized login page or a custom search interface).

Adding an override template

Although the template override system can be used to replace any template on the system, it is most often used to display different types of nodes using different templates. For example, it can be used to display all folders using a template called *my_folder.tpl*, all articles using a template called *my_article.tpl* and so on. The following list of steps demonstrates how eZ publish can be instructed to use a custom template called *my_folder.tpl* whenever a folder node is requested.

1. Create the override rule that will trigger the custom template. It should be added to the *override.ini.append.php* file which belongs to the public (not admin) siteaccess. Example:

```
[my_folder]
Source=node/view/full.tpl
MatchFile=my_folder.tpl
Subdir=templates
Match[class_identifier]=folder
```

The example above will instruct eZ publish to use *my_folder.tpl* instead of *node/view/full.tpl* when a folder node is being viewed using the full view.

2. Add the override template to the design. If the name of the custom design is ”example”, the location of the template should be *design/example/override/templates/my_folder.tpl* (create the directories that do not exist).
3. Add some dummy code to the newly created template (for quick verification). Example:

```
<h1>This is my custom folder template...</h1>

The name of the node is: {$node.name|wash}
```

4. Clear the caches and attempt to view a folder by requesting its URL. The override rule should kick in and the system should display the folder using the newly created template.

Adding a custom system template

This method is typically useful when there is a need to use a modified version of a system template. It does not require the use of override rules; the system template can simply be copied from one of the built-in designs (standard or admin) to the same location, and with the same name, in a custom design. The copied template can then be modified to meet specific needs. The following steps describe the way a custom login template can be added to a design called ”example”.

1. Copy *design/standard/templates/user/login.tpl* to *design/example/templates/user/login.tpl*.
2. For the sake of identifying the custom template, add a line at the top. For example:

```
<h1>This is my custom login template</h1>
```

3. Clear the caches and attempt to access the `login` view of the `user` module (`www.example.com/user/login`); this view makes use of the *login.tpl*. If everything works, the system will detect and use the custom login template.

Including breadcrumbs in the pagelayout

A breadcrumb trail shows a path of links that lead to the current location within a hierarchically structured collection of pages. The path provides an overview of the structure, allowing the user to navigate more easily. Every time a page is rendered, eZ publish automatically creates a data structure containing the path. This structure is stored within the `"path"` component of the `$module_result` array. Depending on the requested URL, the path can either be a trail of nodes leading to the one that is currently being viewed (for example ”Products | Tables | Red kitchen table”) or links to miscellaneous views that lead to the system function currently being accessed (for example ”Content | Search”).

The easiest way to generate breadcrumbs is by including the *page_toppath.tpl* template in the pagelayout:

```
{include uri='design:page_toppath.tpl'}
```

The *page_toppath.tpl* template is part of the standard design. It will be automatically used as long as the system falls back to the standard design; if you want, you can override this template in your custom design (as described in the previous section).

Including a search interface in the pagelayout

Whenever an object is published, the contents of the attributes that are marked "searchable" will be automatically indexed by the search engine. The content module provides two interfaces that can be used for searching: a standard / simple interface and an advanced interface.

A search interface can easily be included by adding a form in the pagelayout (it is possible to have this form in other templates as well). The following example shows the most basic search form which consists of an input field and a button.

```
<form method="get"
        action={'/content/search'|ezurl}>
<input type="text" name="SearchText" />
<input type="submit" name="SearchButton"
        value="Search" />
</form>
```

The form method can be either `GET` or `POST`; the system accepts both types. There are both advantages and disadvantages in either case. For example, if the `GET` method is used, it is possible to bookmark a search because the requested URL contains all the necessary information. On the other hand, the URLs will be long and cryptic.

The search form in the previous example posts data to an interface that simply generates a list of hits. A common feature is to include a link that brings up a more advanced interface which allows the user to fine-tune the search. The following code shows an example of a link to the advanced search interface.

```
<a href={'/content/advancedsearch'|ezurl}>
        Advanced search</a>
```

Changing the search page limit

By default, eZ publish displays only 10 hits per page when showing search results. However, the system is capable of displaying 5, 10, 20, 30 or 50 hits per page. The desired page limit cannot be fed directly to eZ publish, it must be specified using a digit between 1 and 5. The following table shows the available possibilities.

Digit	Page limit
1	5
2	10
3	20
4	30
5	50

According to the table above, a value of 4 will yield a page limit of 30. The following list explains how the page limit can be changed using this value. Note that the change must be done manually inside several templates in order to work properly.

1. Add the following line to the search form in the pagelayout:

```
<input type="hidden" name="SearchPageLimit"
        value="4" />
```

2. Copy the *design/standard/content/search.tpl* template to the custom design.
3. Copy the *design/standard/content/advancedsearch.tpl* template to the custom design.
4. Add the following line to the search form in the newly copied search templates:

```
<input type="hidden" name="SearchPageLimit"
        value="4" />
```

5. Replace all occurrences of the string `"SearchText="` with `"?SearchPageLimit=4&SearchText="` in the newly copied search templates.

Reindexing the search

The search tables are automatically updated every time an object is published. However, sometimes it is necessary to manually update the indexes. A typical case is when several objects are created while some of the attributes are not searchable. What if we want those attributes to be searchable? In this case the class needs to be edited and the *searchable* switch(es) must be enabled. While the missing indexes will be generated when the objects are re-published, a more convenient solution is to run the reindexing script. This script will automatically rebuild the search tables without any interaction. Another advantage of the reindexing script is that it will not create any new versions and / or update the modification dates of the objects. The following example shows how the reindexing script can be run for a siteaccess called ”example”:

```
./update/common/scripts/updatesearchindex.php \
-s example
```

Note that it is possible to use the *--clean* parameter to get rid of existing search data before reindexing.

Including and using a page navigator

Sometimes it is necessary to display information using several pages. For example, a long list of products, search hits, employees, etc. is often broken up into shorter lists. The lists are ”glued” together by an interface that allows the user to jump between the different pages. eZ publish provides an out-of-the-box solution for breaking up long lists by making use of something called a *navigator*. The navigator is a template that takes care of spreading information over several pages. It can be included in any template that generates long lists. The system comes with two navigators; the following table shows their locations.

Name	Type	Location
Simple	Simple	*design/templates/standard/navigator/simple.tpl*
Google	Advanced	*design/templates/standard/navigator/google.tpl*

Note that the ”Google” navigator has nothing to do with the popular search engine. As with all other templates, the navigators may be copied to a custom design and modified. The following example shows how a navigator can be included in order to produce a paged list.

```
{* The desired number of items per page. *}
{def $pageLimit=10}

{* The total number of nodes. *}
{def $nodeCount=fetch( content, list_count,
        hash( parent_node_id, $node.node_id ) )}

{* Fetches a part of the list. *}
{def $nodeArray=fetch( content, list,
        hash( parent_node_id, $node.node_id,
              offset, $view_parameters.offset,
              limit, $pageLimit ) )}

{* Displays the nodes. *}
{foreach $nodeArray as $item}

    <a href={$item.url_alias|ezurl}>
              {$item.name|wash}</a> <br />

{/foreach}

{* The navigator itself. *}
{include name='navigator'
         uri='design:navigator/google.tpl'
         page_uri=$node.url_alias
         item_count=$nodeCount
         view_parameters=$view_parameters
         item_limit=$pageLimit}
```

The following table describes the parameters that must be provided when a navigator is included.

Parameter	Description
name	A custom name space.
uri	The location of the navigator template file.
page_uri	The URL of the current list.
item_count	The number of objects in the list.
view_parameters	The array containing the view parameters.
item_limit	The desired number of items per page.

Resetting the administrator password

In case of an emergency, the administrator password can be reset in several ways. These are the most commonly used methods, starting with the safest and most convenient:

- Log in as another user who has administrator privileges
- Use the `forgotpassword` view of the user module
- Run a database query

Logging in as another user

In some scenarios, an eZ publish site has a couple of users who are placed inside the ”Administrator” user group. This means that these users have administrator privileges and are able to change the password of the administrator user. Log in using one of these users, edit the object that represents the administrator user, change the password and republish the object.

Using the ”forgotpassword” feature

The `user` module provides a feature that can be used to generate a new password for any user account on the system. The password will be sent to the user’s e-mail address. Note that this option can only be used if the `forgotpassword` view has not been disabled. The following example shows how the feature can be used to change the password for the administrator user.

- Access the `forgotpassword` view of the user module:

  `http://www.example.com/user/forgotpassword`
- Type in the e-mail address of the administrator user and press the **Generate new password** button.
- An e-mail will be sent to the administrator’s address. Follow the instructions in the e-mail.

Running a database query

This option requires access to the database. You need to log in to the database engine, select the eZ publish database and run a query. The following example shows how the administrator password can be reset to ”publish” - which is the default password. Note that ”publish” is a weak and commonly known password, it should be avoided because of the obvious security issue.

```
UPDATE ezuser SET password_hash=”publish”,
password_hash_type=5 WHERE contentobject_id=14;
```

The password will be stored in the database without any form for encryption because hash type 5 means ”plain text”. After running the query in the previous example, the password should be changed using the Administration Interface. When the password is changed using the Administration Interface, eZ publish will store an encrypted version of the new password. Note that you can also generate encrypted passwords by running a database query, as demonstrated in the next section.

Changing the username of a user

The username of a user cannot be changed from within the Administration Interface. A username can only be changed by accessing the database and running a query. Note that the passwords in the `ezuser` table depend also on the usernames. This means that when you change the username of a user account, the password must also be updated. The following example shows how to change the username of an account with user ID 113 to ”joshua” and set the password to ”hello”:

```
UPDATE ezuser SET login='joshua',
                  password_hash=MD5('joshua\nhello'),
                  password_hash_type=2
              WHERE contentobject_id=113;
```

Note that an incorrect query might damage your user data. You should always make a backup before attempting to manually change the database.

Adding login functionality

The following template code demonstrates how login functionality can be added to an eZ publish site. The code should be placed inside the pagelayout.

```
{if $current_user.logged_in}
    Logged in: {$current_user.contentobject.name}
    <br />
    <a href={'/user/logout'|ezurl}Logout</a>
{else}
    <a href={'/user/login'|ezurl}Login</a>
{/if}
```

The example above will check if a user is logged in. If yes, it will display the full name of the logged in user along with a **Logout** link. If no user is logged in, a link to the login view will be displayed.

It is possible to force the login view to use a custom pagelayout called *`loginpagelayout.tpl`*. This can be done by setting the `LoginPage` directive within the `[SiteSettings]` block of a configuration override for *`site.ini`* to ”custom” (instead of `"embedded"`).

It is possible to embed the login form in a template. For example, it could be added as a part of the pagelayout, so that it is always visible and accessible). The following example demonstrates how this can be done.

```
<form method="post" action={'/user/login'|ezurl}>

</label>Username:</label>
<input type="text" name="Login" />

</label>Password:</label>
<input type="password" name="Password" />
<input type="submit" name="LoginButton"
        value="Log in" />
</form>
```

The user may be redirected to a specific page after a successful login. This can be done by changing the value of the `DefaultPage` directive within the `[SiteSettings]` block of a configuration override for *`site.ini`*. Another way of doing it is by adding a hidden variable to the login form. The following example demonstrates this by showing how the user can be redirected to "company/internal_affairs".

```
<input type="hidden" name="RedirectURI"
        value="/company/internal_affairs" />
```

Note that the specified URL must not be piped through the `ezurl` operator. The system will automatically generate the correct URL.

Creating a protected area

There is often a need to provide a protected area that can only be accessed by a certain group of users. A typical example could be a company site where most of the content is public except for a section that requires the employees to be logged in. This functionality can be easily achieved by making use of sections and roles. The following example explains how to create a secret section that only users with special privileges can access.

1. Create a folder called "Secret documents".
2. Go to **Setup**, then **Sections**" and create a section called "Secret section".
3. Assign the newly created section to the "Secret documents" folder.
4. Attempt to access the secret documents using the anonymous user; it should not be possible.
5. Go to the **User accounts** tab and create a new user group called "Secret users".
6. Create a new user within the "Secret users" group.
7. Create a new role called "Secret role" and add a new policy to it.
8. When asked which module the policy should grant access to, select **content**. When asked which function the policy should grant access to, select **read**. During the final step, make sure the policy grants access to the "Secret section". Click **OK** (twice).
9. Assign the newly created role to the "Secret users" group.

10. Attempt to log in with the account that was created inside the ”Secret users” group. The user should be able to access the ”Secret documents” part of the site. Anonymous users will still not be able to access this part of the site.

Creating a custom XML tag

Sometimes it is necessary to use custom XML tags that produce special, non-standard output. The following example shows how a custom XML tag called ”alien” can be created. When used, the system will output the contents of the tag wrapped inside a stylesheet. The name of the stylesheet will be specified as a parameter of the custom tag. In addition, the tag will generate a list of child nodes which will contain the children of the node which provides the XML block where the custom tag is used.

1. Create *content.ini.append.php* in *settings/override/* so that the new settings will work for all siteaccesses.
2. Add the following lines to the newly created file:

```
<?php /*

[CustomTagSettings]
AvailableCustomTags[]=alien

*/?>
```

3. Create *alien.tpl* in *design/example/templates/content/datatype/ezxmltags/* (assuming that the name of the design being used is ”example”).
4. Add the following lines to the newly created file:

```
<div class="{$my_class}">
{$content}
</div>

{def $nodeArray=fetch( content, list,
            hash( parent_node_id,
            $#node.node_id ) )}

{foreach $nodeArray as $nodeItem}

    {$nodeItem.object.name|wash} <br />

{/foreach}
```

5. Clear the caches.
6. Attempt to use the newly created custom tag using the following XML syntax:

```
<custom name="alien" my_class="tagrag">
    The quick brown fox jumps over the lazy dog.
</custom>
```

When the contents of the node’s XML `block` is viewed and the `alien` tag is used, the system should include the *alien.tpl* template.

Creating a feedback form

Feedback forms can easily be created using the information collection feature. This mechanism makes it possible to collect input using the native content model of eZ publish. The following example shows how to create a contact form:

1. Create a new class called ”Feedback” consisting of the following attributes:

Name	Identifier	Datatype	Information collector
Name	name	Text line	No
Subject	subject	Text line	Yes
Message	message	Text block	Yes

   The required flag should be set for all attributes.

2. Create an object using the ”Feedback” class in the content root folder. You’ll only need to fill out the ”name” field, for example ”My feedback form”.

3. Create a new template override that will load a custom template called *feedback.tpl* whenever a ”Feedback” object is viewed. The following example shows the lines that should be added to the *override.ini.append.php* of the siteaccess.

```
[feedback]
Source=node/view/full.tpl
MatchFile=feedback.tpl
Subdir=templates
Match[class_identifier]=feedback
```

4. Create the override template (*feedback.tpl*) and place it inside the *templates/override/* directory of the design used by the siteaccess.

5. Add the following code to the newly created template:

```
{include uri='design:content/
        collectedinfo_validation.tpl'}

<h1>{$node.name}</h1>

<form method="post"
        target={'content/action'|ezurl}>

<label>Subject:</label>
{attribute_view_gui
        attribute=$node.object.data_map.subject}

<label>Message:</label>
{attribute_view_gui
        attribute=$node.object.data_map.message}

<input type="submit"
```

```
            name="ActionCollectInformation"
            value="Submit" />
    <input type="hidden" name="ContentNodeID"
            value="{$node.node_id}" />
    <input type="hidden" name="ContentObjectID"
            value="{$node.object.id}" />
    <input type="hidden" name="ViewMode"
            value="full" />

    </form>
```

6. Clear the caches.
7. Attempt to view the feedback node by requesting its URL (`http://www.example.com/my_feedback_form`). The override template will be displayed and it should therefore be possible to submit information to the system. The submitted data will not result in the creation of new objects or nodes, so it will not become a part of the content node tree. However, the information can be viewed and administered using a dedicated interface labeled **Collected information**. This interface can be found in the **Setup** section of the Administration Interface.

Adding a button that creates a new node

The following example shows how a button that triggers the creation of a new object (and a new node) can be added.

```
<form method="post"
        action={'/content/action'|ezurl}>
    <input type="submit" name="NewButton"
            value="Create a new folder" />
    <input type="hidden" name="ClassID"
            value="1" />
    <input type="hidden" name="NodeID"
            value="2" />
</form>
```

It is necessary to tell eZ publish what kind of object / node should be created. In addition, the desired location within the content node tree must be specified. This can be achieved by making use of two hidden input fields: `ClassID` and `NodeID`. The `ClassID` must be the ID number of the class from which you want to create an object. In the example above the class ID is set to "1", which means that the button will create a folder. The `NodeID` must be a valid ID number of an existing node under which the new node should be created (in other words, this number actually denotes the parent node). In the example above, the node ID is set to "2", which means that the new node will be created in the content root folder.

Permissions

Anonymous users will not be able to create new nodes unless the default permission settings are changed. In order to create new nodes, a user needs to have access to the `create` and `edit` functions of the `content` module. Both functions provide several interfaces which make it possible to limit the privileges to specific locations, certain classes and so on.

The `access` fetch function can be used to figure out whether the current user has access (read, edit, create, remove, etc.) to a given content object or content node. The optional parameters *`contentclass_id`* and *`parent_content_class_id`* can be used to limit the results. These parameters are compatible with both class ID numbers and class identifier strings. The *`contentobject`* parameter is compatible with both nodes and objects. The function supports checking for the following access methods:

- bookmark
- create
- edit
- move
- read
- remove
- pdf
- restore
- translate
- versionread

When checking ”create” access, if the *`contentclass_id`* is not specified, the function will return `TRUE` as long as there is a create access for the given object. However, the user might still not be allowed to create the specific class because of other restrictions. The following example demonstrates how to check whether the current user can create a folder below the current node.

```
{def $test=fetch( content, access,
              hash( access, create,
                    contentobject, $node,
                    contentclass_id, 'folder' ) )}
{if $test}
    You can create a folder below
        "{$node.name|wash}".
{else}
    You cannot create a folder below
        "{$node.name|wash}".
{/if}
```

The edit template

Every time a node is created or edited, eZ publish will attempt to load the edit template (*design/example/templates/content/edit.tpl*). If it doesn't find it in the current design and if there are no override rules, the default / standard edit template will be used. The following list explains the process of creating a custom edit template for a class called "Member".

1. Copy the standard edit template to the template override directory of the custom design.
2. Rename the template (for example *member_edit.tpl*).
3. Edit the file; strip away the unnecessary parts and modify it so that it matches the rest of the site.
4. Create a new template override rule that will load a custom template called *member_edit.tpl* when a "member" object is edited. The following example shows the lines that should be added to the *override.ini.append.php* of the siteaccess.

   ```
   [member_edit]
   Source=content/edit.tpl
   MatchFile=member_edit.tpl
   Subdir=templates
   Match[class_identifier]=member
   ```

5. Clear the caches and verify that the custom edit template is used and that it works correctly.

Disabling the discard confirmation dialog

Whenever the user edits something and cancels the process, the system will display a confirmation dialog. In some cases, this dialog may be unnecessary and annoying. It can be easily disabled by adding a hidden variable called `DiscardConfirm` within the form of the edit template. The following example shows how this variable can be used to turn off the confirmation dialog.

```
<input type="hidden" name="DiscardConfirm"
        value="0" />
```

Automatic redirection after editing

It is possible to instruct the system to redirect the user to a specific page when editing is done. This can be achieved by adding a hidden variable called `RedirectAfterPublish` to the form that takes care of creating new nodes. The following example shows how the users can be redirected to "/company/about" after they're done with editing. Note that the specified URL must not be piped through the `ezurl` operator; the system will automatically take care of generating the correct URL.

```
<input type="hidden" name="RedirectAfterPublish"
          value="/company/about" />
```

Adding an edit button

The following template code shows how a button that triggers the editing of an existing node can be added. The example assumes that the code is placed within a node view template (hence the use of the $node variable - if not, the node must be manually fetched and assigned to the variable). Note that both the node ID and the object ID must be provided.

```
<form method="post"
        action={'/content/action'|ezurl}>
    <input type="hidden" name="ContentNodeID"
           value="{$node.node_id}" />
    <input type="hidden" name="ContentObjectID"
           value="{$node.object_id}" />
    <input type="submit" name="EditButton"
           value="Edit" />
</form>
```

The edit action can also be triggered by using a hyperlink instead of a submit button in a form. The link must contain the ID number of the target object. The following example shows how this can be done.

```
<a href={concat( '/content/edit/',
        $node.object.id )|ezurl}>Edit</a>
```

The following example shows how to check whether the current user has sufficient permissions to edit an object.

```
{if $node.object.can_edit}
    {* Display edit button/link. *}
{else}
    {* Display permission denied message. *}
{/if}
```

Adding a remove button

The following template code shows how a button that triggers the removal of an existing node can be added. The example assumes that the code is placed within a node view template (hence the use of the $node variable). Note that both the node ID and object ID must be provided.

```
<form method="post"
        action={'/content/action'|ezurl}>
    <input type="hidden" name="ContentNodeID"
      value="{$node.node_id}" />
    <input type="hidden" name="ContentObjectID"
      value="{$node.object_id}" />
    <input type="submit" name="ActionRemove"
      value="Remove" />
</form>
```

The following example shows how to check whether the current user has sufficient permissions to remove an object.

```
{if $node.object.can_remove}
    {* Display remove button. *}
{else}
    {* Display permission denied message. *}
{/if}
```

Printer-friendly and alternate output

Printer-friendly output may be achieved in several ways. One of the most commonly used methods is to create an alternate, stripped-down version of the pagelayout. This alternate version should not contain elements that would waste paper and ink (for example images, menus, side bars, etc.). The pagelayout can be changed using the `set` view of the `layout` module. This view takes two parameters. The first parameter must be the name of the desired layout (for example, ”print”). The second parameter must be a valid eZ publish URL (either an existing virtual URL or a system URL).

Setting up alternate layouts

Alternate layouts must be specified in a configuration override for *`layout.ini`*. The default configuration defines three alternate layouts: ”fullscreen”, ”popup” and ”print”. The following table shows the names and templates that will be used.

Name	Template
fullscreen	*`fullscreen_pagelayout.tpl`*
popup	*`popup_pagelayout.tpl`*
print	*`print_pagelayout.tpl`*

If eZ publish is instructed to use the ”print” layout to render a page, it will attempt to use a template called *`print_pagelayout.tpl`* instead of the default *`pagelayout.tpl`*. The following example shows how to create a link within the original pagelayout which will generate the page that is currently being viewed wrapped inside *`print_pagelayout.tpl`*.

```
<a href={concat( '/layout/set/print/',
        $requested_uri_string )|ezurl}>
Printerfriendly version
</a>
```

Although the alternate layouts defined in *`layout.ini`* exist in the standard design, it is common practice to create a custom version for the design being used.

Alternate designs

In addition to the alternate layout templates, it is possible to use a different design when an alternate pagelayout is triggered. This can be achieved by using the `SiteDesign` directive within a configuration override for *`layout.ini`*. The following example shows a configura-

tion block that will use *my_layout.tpl* and a design called "tiny". The name of the combination (which will have to be provided as the first parameter to the set view) is "example".

```
[example]
SiteDesign=tiny
PageLayout=my_layout.tpl
```

Note that it is possible to provide alternate layouts using only CSS. However, that is beyond the scope of this book.

PDF export of nodes

It is possible to generate a PDF file based on the contents of a node. This can be achieved by accessing the `pdf` view of the `content` module. The view takes one parameter, which must be the ID number of the target node. The following example shows how to create a link that will generate a PDF version of the node being viewed. Note that the example uses the `$node` variable. This means that the link must either be placed in a node view template or the node must be manually fetched and assigned to the variable.

```
<a href={concat( '/content/pdf/',
      $node.node_id )|ezurl}>
PDF version
</a>
```

PDF generation is based on the same template system used for rendering HTML pages. When a link similar to the one outlined in the example above is requested, eZ publish will use the following template: *design/standard/templates/node/view/pdf.tpl*. As with all other templates, this template can be copied, modified and overridden using the template override system. PDF output heavily depends on a collection of functions that can be accessed using the `pdf` operator. Refer to the online documentation of this operator for more information.

Using the {node_view_gui ...} function

The `node_view_gui` function makes it possible to view a node using the template system. It allows the developer to render a node using a specific view in any template, regardless of the requested URL. This function proves to be very useful in some scenarios. For example, let's say we want to create a big list of nodes where we wish to output the name of the node and an attribute called "properties". The following example show how this could be done.

```
{def $nodeArray=fetch( content, list,
      hash( parent_node_id, 2 ) )}

{foreach $nodeArray as $item}

    {$item.name|wash} -
      {$item.object.data_map.properties.content}

      <hr />

{/foreach}
```

The code can be added to any template without having to create template overrides. But what if there are different types of nodes? A list could contain different products where each product has a dissimilar set of attributes. For example, a Webshop that sells furniture, cars and computers may want to generate an overview that lists the names of all products along with a selected attribute / property. The following table shows an imaginary set of classes for this scenario.

Class	Attributes
Furniture	■ Name ■ Width ■ Height ■ Depth ■ Color ■ Weight (selected) ■ ...
Car	■ Name ■ Make ■ Model ■ Top speed (selected) ■ Weight ■ Color ■ ...
Computer	■ Name ■ Speed ■ Memory (selected) ■ Hard drive ■ Graphics card ■ Sound chip ■ ...

According to the table, when listing furniture, we want to output the weight; when listing cars, we want to output the top speed, and so on. The previous code example will not be able to take care of this unless some sort of conditional checking is introduced. That may be fine for small and simple scenarios, but it will not be an optimal solution. Fortunately, this simple yet very common and sometimes challenging problem can be easily solved by using the `node_view_gui` function and override templates.

We will need to change the loop code, add three override rules and three new templates. The override rules must trigger when certain types of nodes are displayed using a specific view. In this example we will use the `line` view, which means that the override rules will have to trigger when the system attempts to display one of the product nodes using *`.../node/view/line.tpl`*. The following code shows a modified version of the loop from the previous example.

```
{def $nodeArray=fetch( content, list,
      hash( parent_node_id, 2 ) )}
{foreach $nodeArray as $item}
    {node_view_gui content_node=$item view='line'}
    <hr />
{/foreach}
```

The next step is to set up the override rules so that the system will use different templates when the different types of nodes are viewed. The following example shows the necessary lines that should be added to the *override.ini.append.php* of the siteaccess.

```
[furniture_line]
Source=node/view/line.tpl
MatchFile=furniture_line.tpl
Subdir=templates
Match[class_identifier]=furniture

[car_line]
Source=node/view/line.tpl
MatchFile=car_line.tpl
Subdir=templates
Match[class_identifier]=car

[computer_line]
Source=node/view/line.tpl
MatchFile=computer_line.tpl
Subdir=templates
Match[class_identifier]=computer
```

In the final step, the templates are created. According to the rules above, we need to create three override templates:

- *furniture_line.tpl*
- *car_line.tpl*
- *computer_line.tpl*

Each template needs to output the name of the node along with the selected attribute. The following example shows the code for the *furniture_line.tpl* template; it outputs the name of the node and the weight of the furniture.

```
{$node.name|wash} <br />
{attribute_view_gui
        attribute=$node.object.data_map.weight}
```

The *car_line.tpl* template should output the ”top speed” attribute:

```
{$node.name|wash} <br />
{attribute_view_gui
        attribute=$node.object.data_map.top_speed}
```

The *computer_line.tpl* template needs to output the ”memory” attribute:

```
{$node.name|wash} <br />
{attribute_view_gui
        attribute=$node.object.data_map.memory}
```

Once these steps are completed and the cache is cleared, we have a clean, scalable and easy-to-maintain solution for generating a product list. Although this example may seem small and simple, it perfectly demonstrates a flexible concept that may be used to solve both common and intricate template tasks.

Using the ”tip a friend” feature

The `content` module contains a view called `tipafriend`. This view provides an interface that can be used to notify a person through e-mail about content stored in a node. It can be called up by a link which requests the view, along with the ID number of the target node. When accessed, a simple form requesting the user’s name, e-mail address, the recipient’s email address and an optional comment will appear. When the form is submitted, the system will send out an e-mail to the specified address containing an explanation (why, from where and by whom the e-mail was sent) along with a link to the target node. The following list of steps shows how this feature can be used.

1. Create a link which requests the `tipafriend` view along with the ID number of the target node. Example:

```
<a href={concat( '/content/tipafriend/',
        $node.node_id )|ezurl}>
Tip a friend
</a>
```

2. Optional: copy the *`design/standard/templates/content/tipafriend.tpl`* file to the design being used. This template contains both the form and the messages that appear when the form is submitted. Modify the template so that it matches the site.
3. Optional: copy the *`design/standard/templates/content/tipafriendmail.tpl`* file to the design being used. This template dictates the overall look of the tipa-friend e-mails that will be sent out by the system. Modify the template to suit your needs.
4. Clear the caches and test the solution.

Wrapping PHP functions

It is possible to wrap simple PHP functions and use them in templates without having to code an actual template operator. The system allows direct wrapping of functions that require either one parameter or no parameters. In other words, it is not possible to wrap PHP functions that depend on input based on more than one parameter. Function wrapping must be set up in a configuration override for *`template.ini`*. The following example shows a setup that introduces a new template operator called ”myoperator”, which will be a wrapper for soundex, a PHP function that calculates the soundex key of a string.

```
[PHP]
PHPOperatorList[myoperator]=soundex
```

The PHP `soundex` function expects a string as a parameter. Its job is to calculate the soundex key of a word. Words that are pronounced similarly produce the same key. Soundex keys can,

for example, be used to simplify searches in cases where the user doesn't remember the correct spelling of a word. The PHP function returns a four-character string that starts with a letter and ends with digits. Note that soundex keys are optimized for English pronunciation. The following example shows the usage of the newly introduced template operator in a template:

```
{'Knuth'|myoperator}
```

The example above will return "K350", which is the soundex key for "Knuth".

Custom HTTP meta tags

Although the specification of meta tags does not define a set of legal meta data properties, it is a common practice to include generic information such as the name of the author, description of the site, copyright notices, keywords, etc. This can be done by using the `MetaDataArray[...]` directive in a configuration override for *`site.ini`*. The configuration is usually done on a siteaccess basis.

Good meta information leads to higher page ranking, yielding higher visibility with search engines. eZ publish will output the name and value of the specified tags when the standard *`page_head.tpl`* template is included in the pagelayout. If no custom tags are defined, the default settings will be used. The following example shows a meta tag setup for an imaginary site.

```
[SiteSettings]
MetaDataArray[author]=Miles Bennet Dyson
MetaDataArray[copyright]=Cyberdyne Systems
MetaDataArray[description]=Artificial Intelligence
MetaDataArray[keywords]=artificial intelligence,
        neural networks
...
```

Disabling access to modules and views

A typical eZ publish site uses only a small portion of all the functionality provided by the system. One of the most basic and effective ways of preventing possible attacks, misuse, information leakage, etc. is to disable unused modules and / or views. The more information a perpetrator is able to collect, the more likely it is that the site will be compromised. For example, the `about` view of the `ezinfo` module reveals that the web site is running eZ publish, along with the version number. This information is available for all visitors. Someone may find it handy to inject a possible exploit that works with the unpatched version which the site is currently running.

Disabling functionality is typically one of the last steps made before going live. The following example shows how different modules and views can be disabled from within the *`site.ini.append.php`* configuration file of a siteaccess.

```
[SiteAccessRules]
Rules[]=access;disable
Rules[]=module;class
```

```
Rules[]=module;ezinfo
Rules[]=module;form
Rules[]=module;pdf
Rules[]=module;content/tipafriend
etc.
```

The example above disables access to the following modules: `class`, `ezinfo`, `form` and `pdf`. In addition, the last line demonstrates how access to a single view can be disabled (in this case, the `tipafriend` view of the `content` module). Access to the modules in the example above (and many others) is typically not needed for a public site. Although eZ publish is built with security in mind and access is controlled by the permission system, it is still a good idea to totally close down unused modules - better to be safe than sorry.

Debugging a live site

Sometimes it is necessary to debug a site while it is being used by hundreds of visitors. Turning on the debug output while the site is live may not be smart. It can disrupt a nice design, scare the visitors (and owners) and it may even reveal sensitive data that might be used by a potential cracker. Fortunately it is possible to instruct eZ publish to only display the debug output if the visitor is coming from a specific IP address. The following example shows how IP based debug output can be used to debug a live site from a computer (or a gateway/NAT) behind ”62.70.12.230”. The lines must be added to the configuration override of *`site.ini`* for the public siteaccess.

```
[DebugSettings]
DebugOutput=enabled
Debug=inline
DebugByIP=enabled
DebugIPList[]
DebugIPList[]=62.70.12.230
```

Note that it is possible to add several IP addresses. This can be done by providing more `DebugIPList[]=xxx.yyy.zzz.www` lines in the configuration file. Be aware that the system will spend more time generating a page that contains debug information. However, if debugging is limited to a small set of IP addresses, it will not slow down every page, only the ones requested by hosts hiding behind the specified IPs. In other words, it is safe to leave the master debug switch on as long as the debug-by-ip feature is used.

Backing up and restoring a site

An eZ publish site stores information using two separate storage solutions: a database and the filesystem. It is important to understand that a backup must include both a dump of the database and a copy of the eZ publish directory. If either one of them is missing, the backup will most likely be useless (worst case scenario).

A common way of creating a backup is by dumping the database to a file inside the eZ publish directory, then creating a compressed version of the entire thing. It is a good idea to exclude the

cache files. Including them will not only result in unnecessarily large backup files, it will also significantly slow down the backup process. Copying the files without doing any packaging or compression is not a good idea. This is because eZ publish consists of a big and deep directory tree with long filenames. Some filesystems (for example, ISO9660 for compact discs) are unable to store the structure, and hence the backup could be rendered useless.

Dumping the database

The following example shows how a database called ”example” can be dumped to a file called *backup.sql* using a command line tool that comes with MySQL.

```
$ mysqldump -u root --add-drop-table \
  example > backup.sql
```

Note that the username ”root” may have to be changed to the name of the user which has access to the eZ publish database. The *--add-drop-table* parameter will add some extra queries at the beginning of the dump. These statements will get rid of all tables when the dump is put back into a server where the database already exists. This will prevent problems with conflicting tables (no ”table already exists” messages will be displayed) because the script starts by dropping the existing tables before it attempts to create new ones.

Backing up the eZ publish directory

As mentioned earlier, the eZ publish directory is huge. Relying on raw copies of it is not considered to be a safe approach. A better solution is to create an archived, compressed version of it (for example tar, zip, rar, etc.). The following example shows how an eZ publish directory called ”my_site” can be archived and compressed into a single file called *backup.tar.gz* using the **tar** utility on a Linux / UNIX system.

```
$ tar zcf backup.tar.gz my_site
```

Restoring an archived eZ publish directory

The previous example showed how an eZ publish directory could be archived. This is how the archive (called *backup.tar.gz*) can be unpacked and restored:

```
$ tar zxf backup.tar.gz
```

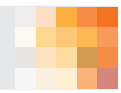
The `verbose` parameter has been deliberately omitted from the above example. When the verbose option is used, the console will be filled with hundreds of lines showing the files that are being processed. This can make it difficult to notice and isolate individual issues.

Restoring an eZ publish database

The following example shows how a database dump called *backup.sql* can be restored to a database called ”example” using one of the MySQL command line tools.

```
$ mysql -u root example < backup.sql
```

Chapter 5. Extensions



This chapter provides an introduction to extending eZ publish. It is targeted at developers who want to add custom functionality to the system. Because extensions are written in the PHP programming language, some familiarity with programming concepts and practices is required. After reading this chapter, a moderately experienced programmer will understand the potential of the extension system, and should be able to create eZ publish extensions. The following topics are covered:

- The purpose and concepts of the extension system
- The extension directory structure
- How to enable and disable extensions
- How to create a design extension
- How to create a custom datatype
- How to create a custom template operator
- How to create a custom workflow event

Extension overview

The eZ publish architecture is designed to support the use of custom code that extends the functionality of the system without modifying the original distribution. Separating the core and custom functionality makes the system easy to upgrade and maintain and enables programmers to share their work with other eZ publish users without having to redistribute the entire system.

Custom code, design, translations, etc. are added via extensions. An *extension* is a set of files containing PHP code (and other custom components). The extension system can be used to add custom components such as:

- Datatypes
- Designs
- Workflow events
- Modules
- Content actions
- Translations
- Template operators

Extensions are used whenever there is a requirement to add custom functionality to eZ publish. For example, if special data for which there is no built-in datatype needs to be stored, a new datatype should be added as an extension. As another example, a custom workflow event could be created that notifies an external system when an object is published. Extensions may also be used to override existing eZ publish functionality, for example to tailor an aspect of the system to your exact requirements. Regardless of the nature of the extension, the primary advantage of extensions is that none of the original eZ publish files are modified. All custom code is completely separated from the rest of the system.

Directory structure

All the files that make up an extension are stored in the *extension/* directory within the eZ publish installation. There is no limit on the number of extensions that can be used with eZ publish. Within the *extension/* directory, each individual extension is stored in its own subdirectory. The name of a subdirectory functions as the name of an extension. The following example shows three extensions within the main extension directory: ”myExtension”, ”yourExtension” and ”anotherExtension”.

```
ezpublish
|
+--extension
   |
   +--myExtension
   |
   +--yourExtension
   |
   +--anotherExtension
```

An extension’s directory can contain the following subdirectories:

Subdirectory	Description
actions/	Actions for forms.
datatypes/	Datatypes.
design/	Designs (design-related files).
eventtypes/	Workflow events.
modules/	Modules (modules, views, fetch functions, etc.).
settings/	Configuration settings that belong to the extension. This directory is required unless the extension only provides a template operator.
translations/	Translations.

Extension activation

After installing an extension, the site must be configured to use it. This configuration can be done on a siteaccess or on a global basis. (Unless there are specific reasons to do otherwise, the most common practice is to allow all siteaccesses to use all extensions.) The following example shows the configuration override within *site.ini.append.php*.

```
[ExtensionSettings]
ActiveExtensions[]=myextension
ActiveExtensions[]=yourextension
```

These lines instruct the system to use two extensions called ”myextension” and ”yourextension”.

Design extensions

The design directory in an extension that can contain:

- Custom design for a site
- Design related to the extension itself (workflow templates, datatype templates, etc.)
- Additional design for an existing design (such as replacement templates for the standard, admin and other designs)

The structure of an extension’s design directory follows the same layout as the default design directory in the eZ publish installation. The design directory must contain a subdirectory for each design that it provides. The names of the subdirectories are used as the design names. The design subdirectories may contain images, templates, stylesheets, etc., in the same manner as a design in the main design directory.

As mentioned above, you can create an extension for an existing design. For example, the directory structure for an extension that extends the ”standard” design would be: *`extension/example/design/standard/`* (assuming that the name of the extension itself is ”example”).

Creating a design extension

The following steps show how to create a design extension. The name of the extension will be ”myextension” and the name of the design will be ”example”. Note that the designs provided by an extension will be ignored unless the system is instructed to use them (as described in the ”Extension Activation” section above).

1. Create a subdirectory called *`myextension/`* in the *extension/* directory.
2. Create the extension’s directory structure:
   - Create a subdirectory called *`design/`* in the *`myextension/`* directory. This directory will contain all the designs provided by the extension.
   - Create a subdirectory called *`example/`* in the *`design/`* directory. The structure of the *example/* directory must follow the standard design subdirectory structure (it can ave subdirectories containing templates, CSS files, images, etc.).
   - Create a subdirectory called *`settings/`* in the *`myextension/`* directory. This directory will contain the settings for the ”myextension” extension.
3. Create a file called *`design.ini.append.php`* in the *`settings/`* directory containing the following lines:

   ```
   <?php /*
   ```

```
[ExtensionSettings]
DesignExtensions[]=myextension

*/ ?>
```

This instructs eZ publish to use the designs provided by ”myextension”. The designs will be enabled when the extension itself is activated.

4. Activate the extension either globally or for a siteaccess by adding the following lines to a configuration override for *site.ini*:

```
[ExtensionSettings]
ActiveExtensions[]=myextension
```

Datatype extension

Although the default datatypes are sufficient for most sites, custom datatypes are one of the most powerful extensions that can be added to eZ publish. They make it possible to store data differently than the built-in datatypes and will thus extend the content management capabilities of the system. One of the most important properties of a datatype is the graphical user interface and the validation logic that is used during input.

Whenever you need to store and validate information in a way that would be cumbersome using one of the built-in datatypes, a custom datatype should be created. For example, if you need to validate and store IP addresses, instead of using the built-in ”Text line” datatype you should develop a custom datatype that provides four input fields and data validation. This enhances the input process and eliminates the risk of storing invalid information.

Creating a new datatype

To create an extension called ”mydatatype”:

1. Create a directory called *mydatatype/* beneath the *extension/* directory.
2. Create the extension’s directory structure:
   - Create a subdirectory called *settings/* in the *mydatatype/* directory. This directory will contain the settings for the ”mydatatype” extension.
   - Create a subdirectory called *design/* in the *mydatatype/* directory. This directory will contain a design for the datatype’s templates.
   - Create a subdirectory called *datatypes/* in the *mydatatype/* directory. This will store the datatypes provided by the extension.
   - Create a subdirectory called *ezexample/* in the *datatypes/* directory. This will contain the PHP code for the datatype.
3. Copy an existing datatype’s PHP file into the *ezexample/* directory. For example, copy *kernel/classes/datatypes/ezisbn/ezisbntype.php*. Rename the file to *ezexampletype.php*. The filename must end with *type.php*. (Note that you can use the rapid application development (RAD) tools in the Administration Interface to generate a framework for a new datatype.)

4. Create program code for the new datatype in *ezexampletype.php*.
5. Create the view and edit templates for the datatype. These are required. Example: *extension/mydatatype/design/standard/templates/content/datatype/view/ezexample.tpl* and *extension/mydatatype/design/standard/templates/ content/datatype/edit/ezexample.tpl*.
6. Tell eZ publish that it should use the designs that are provided by the extension. Do this by creating a *design.ini.append.php* file in the *settings/* directory. Make sure that it contains the following lines:

```
<?php /*

[ExtensionSettings]
DesignExtensions[]=mydatatype

*/ ?>
```

7. Tell eZ publish that it should use the datatype that is provided by the extension. Do this by creating a file called *content.ini.append.php* in the *settings/* directory. Make sure that it contains the following lines:

```
<?php /*

[DataTypeSettings]
ExtensionDirectories[]=mydatatype
AvailableDatatypes[]=ezexample

*/ ?>
```

8. Activate the extension for a siteaccess or globally by adding the following lines to a configuration override for *site.ini*:

```
[ExtensionSettings]
ActiveExtensions[]=mydatatype
```

Programming the datatype

The following example shows the framework for a datatype.

```
<?php
include_once( '/kernel/classes/ezdatatype.php' );
define( 'EZ_DATATYPESTRING_EXAMPLE', 'ezexample' );
class eZExampleType extends eZDatatype
{
...
}
eZDataType::register( EZ_DATATYPESTRING_EXAMPLE,
        'ezexampletype' );
?>
```

The example shows the PHP file which must take care of the following:

- Include the file containing the superclass.
- Define a unique ID for the datatype.
- Provide a class that extends `eZDataType`.
- Register the datatype.

Functions

At a minimum, the following set of functions should be provided:

- A constructor
- `validateObjectAttributeHTTPInput(...)`
- `fetchObjectAttributeHTTPInput(...)`
- `storeObjectAttribute(...)`
- `objectAttributeContent(...)`
- `hasObjectAttributeContent(...)`
- `isIndexable(...)`
- `metaData(...)`
- `title(...)`

The constructor

The constructor must run the constructor of the superclass using the unique ID and the actual name of the datatype as parameters. The name will be used in various interfaces in the Administration Interface. In addition, the constructor must reveal whether the datatype supports serialization or not. A datatype that supports serialization will allow export and import of a class or of actual content that uses the datatype. (The serialization map must also be provided.)

validateObjectAttributeHTTPInput(...)

This function validates the input data. For example, if the datatype is used for storing credit card numbers, this function must determine whether the number provided is valid. If the data is valid, the function must return EZ_INPUT_VALIDATOR_STATE_ACCEPTED. If the data is invalid, the function must return EZ_INPUT_VALIDATOR_STATE_INVALID along with a validation error message.

The *`http`* parameter holds the class object `eZHTTPTool`, which can be used to extract and check the posted input. The *`base`* parameter holds the base name of the HTTP variable (usually *`ContentObjectAttribute`*). The *`contentObjectAttribute`* parameter holds the attribute object itself.

fetchObjectAttributeHTTPInput(...)

This function must get the input data and store it in the data instance. The data may be stored either as text (`data_text`), integer (`data_int`) or a floating-point value (data_float). The parameters are the same as for the `validateObjectAttributeHTTPInput(...)` function.

storeObjectAttributeHTTPInput(...)

This function must be used if `fetchObjectAttributeHTTPInput(...)` does not fully handle storing the input data.

objectAttributeContent(...)

This function must return the content stored by the datatype. The *`contentObjectAttribute`* parameter gives access to the attribute object.

hasObjectAttributeContent(...)

This function must return either `TRUE` or `FALSE` depending on whether the datatype has stored any content or not. The *`contentObjectAttribute`* parameter gives access to the attribute object.

isIndexable(...)

This function must return either `TRUE` or `FALSE` depending on whether the datatype supports search indexing or not.

metaData(...)

This function must return the metadata that should be used when the attribute is being indexed by the search engine.

title(...)

This function must generate and return a short string which is based on the input data. The string will be used to set the object name if the object's name pattern includes an attribute that uses the datatype.

Class attributes

If the datatype handles class attribute definitions (for example, a default value, input constraints, etc.), it must provide additional functions and two templates: one for class editing and one for class viewing. One or more of the following functions must be implemented:

- `storeClassAttribute(...)`
- `validateClassAttributeHTTPInput(...)`
- `fixupClassAttributeHTTPInput(...)`
- `fetchClassAttributeHTTPInput(...)`

Refer to the superclass for more information about these functions. The class templates must be placed in the *`templates/class/datatypes/edit/`* and *`templates/class/datatypes/view/`* directories of the design provided by the extension.

Template operator extension

The built-in template operators offer a wide range of functionality. While these operators are sufficient for common template tasks, custom template operator extensions are required for advanced purposes. For example, a template operator extension could be created for special text

processing tasks. In such cases, the best solution is to create a custom template operator as an extension. Custom template operators function in the same way as the builtin operators. Once a custom operator is added, it can be used from within any template (assuming that the operator is available for all siteaccesses). The following section describes how to create template operator extensions.

Creating new template operators

To create a template operator extension called ”myoperators”:

1. Create a directory called *myoperators/* in the *extension/* directory.
2. Create the extension’s directory structure:
   - Create a subdirectory called *settings/* in the *myoperators/* directory. This directory will contain the settings for the ”myoperators” extension.
   - Create a subdirectory called *autoloads/* in the *myoperators/* directory. This directory will contain the code for the operators provided by the extension.
3. Create a new file called *eztemplateautoload.php* inside the *autoloads/* directory. Make sure that it contains the following lines:

```
<?php
$eZTemplateOperatorArray = array();
$eZTemplateOperatorArray[] =
  array( 'script' => 'extension/
          myoperators/autoloads/myoperators.php',
         'class' => 'MyOperators',
         'operator_names' => array
                 ( 'operatorOne',
                   'operatorTwo' ) );
?>
```

   This file informs eZ publish about the names and the locations of the new operators. In this example we have two operators called ”operatorOne” and ”operatorTwo”. They will be defined in a class called ”MyOperators” located in *myoperators.php*.

4. Create a new file inside the *autoloads/* directory that will contain the operator logic. In this example, we will create a file called *myoperators.php*. Enter code for the new operator(s). (Refer to the next section for more information.)
5. Create a *site.ini.append.php* file in the *settings/* directory of the extension. This tells eZ publish that it should use the operator(s) provided by the extension. The file should contain the following lines:

```
<?php /*
[TemplateSettings]
ExtensionAutoloadPath[]=myoperators
*/ ?>
```

6. Activate the extension for a siteaccess or globally by adding the following lines to a configuration override for *site.ini*:

```
[ExtensionSettings]
ActiveExtensions[]=myoperators
```

Programming the operators

The code for the operators must be stored in *myoperators.php* (this is the file that was specified in *eztemplateautoload.php*). The following example shows how the skeleton code for the two operators (”operatorOne”) and (”operatorTwo”) should be set up.

```
<?php

class MyOperators
{
    // Constructor.
    function MyOperators()
    {
        $this->Operators = array
                ( 'operatorOne', 'operatorTwo' );
    }

    // Returns the operators that
    // are provided by this class.
    function &operatorList()
    {
        return $this->Operators;
    }

    // This is needed for classes that
    // provide several operators.
    function namedParameterPerOperator()
    {
        return true;
    }
    // Returns an array of the named parameters.
    function namedParameterList()
    {
        return array( 'operatorOne' => array(),
                      'operatorTwo' => array() );
    }

    // This function provides the main logic.
    function modify( &$tpl,
                     &$operatorName,
                     &$operatorParameters,
                     &$rootNamespace,
                     &$currentNamespace,
```

```
                    &$operatorValue,
                    &$namedParameters )
    {
        switch ( $operatorName )
        {
            case 'operatorOne':
            {
                $operatorValue = $this->doSomething
                        ( $operatorValue );
            } break;
            case 'operatorTwo':
            {
               $operatorValue = $this->
                       doSomethingElse
                               ( $operatorValue );
            } break;
       }
   }
   // Custom function #1, used by "operatorOne".
   function doSomething( $input )
   {
       // Process the input and
       // return some output.
   }
   // Custom function #2, used by "operatorTwo".
   function doSomethingElse( $input )
   {
       // Process the input and
       // return some output.
   }
   var $Operators;
}
?>
```

Note that this example only shows how the framework should be set up. You'll have to add additional code to implement the desired functionality.

Parameter handling

The operators in the previous example do not support any parameters except the input parameter provided through a pipe. Suppose that "operatorOne" needed two additional parameters: the first parameter is required and the default value of the second parameter is "MP3". To implement these parameters, we change the `namedParameterList` function:

```
function namedParameterList()
{
```

```
    return array(
        'operatorOne' => array(
            'first_param' =>
                   array( 'type' => 'string',
                          'required => 'true',
                          'default' => '' ),

            'second_param' =>
                    array( 'type' => 'string',
                           'required => 'false',
                           'default' => 'MP3' ) ),

        'operatorTwo' => array() );
}
```

The parameters will be available in the modify function and can be accessed in the following way:

```
$one = $namedParameters['first_param'];
$two = $namedParameters['second_param'];
```

These parameters should of course be sent to the function that contains the code for the ”operatorOne” operator, in this case the ”doSomething(...)” function.

Workflow extensions

Sometimes it is useful to create custom workflow events that perform miscellaneous tasks. The tasks may be either interactive or non-interactive. Custom events can, for example, be used to manipulate information stored in the content node tree, to communicate with remote servers, to process user input, to carry out maintenance tasks, and so on.

A workflow consists of one or more events. The events are processed in the sequence they were defined when the workflow was created. A typical event executes some code and then terminates, allowing the next event in the sequence to be processed. This is the most usual case. However, an event may repeat itself, block the workflow while waiting for something to happen or even abort the entire workflow. The behavior of an event is controlled by a status code.

Status codes

Regardless of what an event does, it must always return a status code at the end. The status code determines whether the event should be repeated, show a template, redirect the user, abort the entire workflow or allow the next event to be processed. The following table shows the available status codes.

Status	Description
EZ_WORKFLOW_TYPE_STATUS _ACCEPTED	This status can be returned when an event has finished and there is nothing more to be done. The system will process the next event of the workflow. If no other events exist, the workflow will finish.

Status	Description
EZ_WORKFLOW_TYPE_STATUS_REJECTED	If an unexpected error occurs, an event may terminate its own execution by returning this status. Currently it is equal to EZ_WORKFLOW_TYPE_STATUS_WORKFLOW_CANCELLED. The result is that the entire workflow will be terminated and the final status will be CANCELLED.
EZ_WORKFLOW_TYPE_STATUS_WORKFLOW_CANCELLED	An event may terminate the entire workflow by returning this status. The workflow will stop and no other events will be processed.
EZ_WORKFLOW_TYPE_STATUS_WORKFLOW_DONE	This status allows an event to stop a workflow as if it was completely finished. In other words, it fools the system into believing that the workflow has completed without any problems and thus the operation (which triggered the workflow) will continue normally.
EZ_WORKFLOW_TYPE_STATUS_DEFERRED_TO_CRON	This status will cause the workflow to be deferred to the cronjob. When the cronjob executes, the system will process the next event of the workflow. If no other events exist, the workflow will finish.
EZ_WORKFLOW_TYPE_STATUS_DEFERRED_TO_CRON_REPEAT	This status will cause the workflow to be deferred to the cronjob. When the cronjob executes, the system will process the same event which deferred the workflow. In other words, this status can be used to repeat an event until some condition is matched.
EZ_WORKFLOW_TYPE_STATUS_FETCH_TEMPLATE	This status will instruct the system to stop processing the workflow and show a template. The template should implement a form containing a submit button. When used, it will allow the workflow to continue. If the template does not provide a form, the workflow will still continue the next time it is triggered. In both cases, eZ publish will start processing the next event of the workflow. If no other events exist, the workflow will finish.
EZ_WORKFLOW_TYPE_STATUS_FETCH_TEMPLATE_REPEAT	This status will result in almost the same behavior as EZ_WORKFLOW_TYPE_STATUS_FETCH_TEMPLATE. The only difference is that when the workflow continues, the system will process the very same event which returned the status. In other words, this status can be used to keep displaying a template until a specific condition occurs.
EZ_WORKFLOW_TYPE_STATUS_REDIRECT	This status will instruct the system to stop processing the workflow and redirect the user to a specified location. For example, it can be used when the workflow is triggered in a checkout process where the user needs to be redirected to an external payment solution. When finished, the user is then redirected back to the checkout process and the next event will be processed. If no other events exist, the workflow will finish.
EZ_WORKFLOW_TYPE_STATUS_REDIRECT_REPEAT	This status will result in almost the same behavior as EZ_WORKFLOW_TYPE_STATUS _REDIRECT. The only difference is that when the workflow continues, the system will process the same event which returned the status. In other words, this status can be used to keep redirecting a user until a specific condition occurs.

Creating a custom event

To create a custom event via an extension called ”myevent”:

1. Create a new directory called *myevent/* in the *extension/* directory.
2. Create the extension’s directory structure:
   - Create a subdirectory called *settings/* in the *myevent/* directory. This directory will contain the settings for the ”myevent” extension.
   - Create a subdirectory called *design/* in the *myevent/* directory. This directory will contain a design that provides the workflow event’s templates.
   - Create a subdirectory called *eventtypes/* in the *myevent/* directory. Inside *eventtypes/* create a directory called *event/*.
   - Create a subdirectory called *ezexample/* in the *event/* directory. This will store the PHP code for the workflow event.
3. Enter program code for the new workflow event in *ezexampletype.php* (described in the next section). The file must be stored in the *ezexample/* directory.
4. Optional: create the result template for the event if it uses a template. For example: *extension/myevent/design/standard/templates/workflow/eventtype/result/event_ezexample.tpl*
5. Optional: if the event makes use of a template (located within the same extension), create a *design.ini.append.php* file in the *settings/* directory of the extension. This will make eZ publish use the designs that are provided by the extension. Ensure that it contains the following lines:

```
<?php /*
[ExtensionSettings]
DesignExtensions[]=myevent
*/ ?>
```

6. To configure eZ publish to use the workflow event provided by the extension, create a file called *workflow.ini.append.php* in the *settings/* directory of the extension. Ensure that it contains the following lines:

```
<?php /*
[EventSettings]
ExtensionDirectories[]=myevent
AvailableEventTypes[]=event_ezexample
*/ ?>
```

7. Activate the extension for a siteaccess or globally by adding the following lines to a configuration override for *site.ini*:

```
[ExtensionSettings]
ActiveExtensions[]=myevent
```

Programming the event

The following example shows the framework for workflow events. This event does nothing useful, it is used merely to demonstrate how an event should be programmed.

```
<?php

define( 'EZ_WORKFLOW_TYPE_EXAMPLE_ID', 'ezexample' );

class eZExampleType extends eZWorkflowEventType
{
    function eZExampleType()
    {
        $this->eZWorkflowEventType
                ( EZ_WORKFLOW_TYPE_EXAMPLE_ID,
                        'Example' );
        $this->setTriggerTypes( array
                ( 'content' => array( 'read' =>
                        array ( 'before' ) ) ) );
    }
    function execute( &$process, &$event )
    {
        return EZ_WORKFLOW_TYPE_STATUS_ACCEPTED;
    }
}

eZWorkflowEventType::registerType
       ( EZ_WORKFLOW_TYPE_EXAMPLE_ID,
              'ezexampletype' );

?>
```

As the example shows, the event starts by defining a unique ID. Every event must provide a class which extends `eZWorkflowEventType`. The class must contain two functions: a constructor and the actual process function. The constructor must take care of two things:

- Run the constructor of the superclass using the event's unique ID and name as parameters.
- Show which trigger / operation types the event supports.

In the example above, the event will be made compatible with the "before" trigger of the `read` function in the `content` module. In other words, it will be possible to trigger this event before a node is viewed. The system uses this information to determine which workflows can be connected to the different triggers. If a workflow starts with an event that is compatible with a specific trigger, it will automatically pop up in the dropdown list for that trigger in the Administration Interface.

Note that the default *`workflow.ini`* file does not enable all connection types. For example, the "content/read" operation must be added manually if it is to be used. This is controlled by the `AvailableOperationList` directive of `[OperationSettings]`. The operations for the `content` module are defined inside *`kernel/content/operation_definition.php`*. You'll need to create *`settings/override/workflow.ini.append.php`* and containing the following lines:

```
<?php /*

[OperationSettings]
```

```
AvailableOperationList[]=content_read
*/ ?>
```

The `execute` function is the main processing function. This is where the event carries out its tasks. The function gives access to two variables: `process` and `event`. While `process` contains an `ezworkflowprocess` object, the `event` variable contains an `ezworkflow-event` object. These objects can be used to extract and update important information when the event is running. For example, it is possible to get the ID number of the current node, object, user and so on.

Regardless of what it does, the `execute` function must always return a valid status code. The example above returns the accepted status, which means that the workflow will continue executing the next event in the chain.

An event must register itself using the `registerType` function of the `eZWorkflowEvent-Type` class. This must be done at the very end of the event's PHP file.

Event example

The following example shows a simple event. It will keep displaying a template until the user clicks "Continue". This template could for example contain an advertisement. The event should be added to a workflow that is connected to the "content/read/before" operation. It will fetch the node that is about to be displayed and make it available in the event's template.

```
<?php
define( 'EZ_WORKFLOW_TYPE_EXAMPLE_ID', 'ezexample' );
class eZExampleType extends eZWorkflowEventType
{
    function eZExampleType()
    {
        $this->eZWorkflowEventType
                ( EZ_WORKFLOW_TYPE_EXAMPLE_ID,
                        'Example' );
        $this->setTriggerTypes( array
                ( 'content' => array( 'read' =>
                        array ( 'before' ) ) ) );
    }
    function execute( &$process, &$event )
    {
       $parameters =& $process->attribute
               ( 'parameter_list' );
       $http =& eZHTTPTool::instance();

       if( $http->hasPostVariable
               ( 'ContinueButton' ) )
       {
           return EZ_WORKFLOW_TYPE_STATUS_ACCEPTED;
```

```
        }

        $node =& eZContentObjectTreeNode::fetch
                ( $parameters['node_id'] );

        $requestUri = eZSys::requestUri();

        $process->Template = array( 'templateName' =>
                 'design:workflow/eventtype/result/
                         event_ezexample.tpl',
                                     'templateVars' =>
                 array( 'node' => $node,
                         'request_uri' =>
                                 $requestUri ) );

        return EZ_WORKFLOW_TYPE_STATUS_FETCH_
                 TEMPLATE_REPEAT;
    }
}

eZWorkflowEventType::registerType(
        EZ_WORKFLOW_TYPE_EXAMPLE_ID,
                 'ezexampletype' );

?>
```

The example event's template will output the name of the node that is about to be displayed. In addition, it will also provide the form that makes it possible to post an action in order to continue the workflow. The following example shows the event's template code (the contents of the *extension/myevent/design/standard/templates/workflow/eventtype/result/event_ezexample.tpl* file).

```
<h1>Stop!</h1>

You are about to view "{$node.name|wash}"...

<form method="post" action="{$request_uri|ezurl}">
<input type="submit" name="ContinueButton"
           value="Continue" />
</form>
```

To implement the extension, clear the caches and create a new workflow using the Administration Interface. Add the custom event (which should now be available in the dropdown list). Enable the "content_read" trigger by adding a global override for *workflow.ini* (as described earlier) and have it start the newly created workflow on "content/read/before" (from within **Triggers** under **Setup** in the Administration Interface).

Test the event by viewing a node. Instead of displaying the node, the system should now run the workflow which in turn executes the event. The event should show the template containing the **Continue** button. Since there are no more events in the workflow, it will self-terminate when the button is clicked and thus the system will show the node that was requested.

Appendix A. Appendix

Datatypes

Datatype	Summary	Searchable	Collector
Authors	Stores info about additional authors.	Yes	No
Checkbox	Stores a binary value (on or off).	Yes	Yes
Date	Validates and stores a date value.	Yes	No
Date and time	Validates and stores a date and a time value.	Yes	No
E-mail	Validates and stores an email address.	Yes	Yes
File	Stores any type of file.	Yes	No
Float	Validates and stores a decimal value.	No	No
Identifier	Generates a non-editable identification string.	Yes	No
Image	Validates and stores a digital image.	No	No
Integer	Validates and stores an integer value.	Yes	No
ISBN	Validates and stores an ISBN value.	Yes	No
Keywords	Stores keywords.	Yes	No
Matrix	Stores multiple rows and columns of text.	Yes	No
Media	Stores a media file (Flash/QuickTime/Real/etc.).	No	No
Multi-option	Allows option selections. [Webshop]	Yes	No
Object relation	Stores a relation to a content object.	Yes	No
Object relations	Stores relations to other content objects.	Yes	No
Option	Allows an option selection. [Webshop]	No	Yes
Price	Stores a price (including / excluding VAT). [Webshop]	Yes	No
Range option	Allows an integer selection. [Webshop]	Yes	No
Selection	Stores single and multiple choices.	Yes	No
Text block	Stores multiple lines of unformatted text.	Yes	Yes
Text line	Stores a single line of unformatted text.	Yes	Yes
Time	Validates and stores a time value.	No	No
URL	Validates and stores a URL / address.	No	No
User account	Validates and stores info about a user.	Yes	No
XML block	Validates and stores multiple lines of formatted text.	Yes	No

Note that information collection support has been added to even more datatypes in eZ publish 3.8.

Modules

Module	Description
`class`	Provides views for managing classes, class groups, etc.
`collaboration`	Provides an interface to the collaboration engine.
`content`	Provides views for managing content (nodes, objects, searching, etc.)
`error`	Provides an interface for error handling / reporting.
`ezinfo`	Provides views for displaying information about eZ publish.
`form`	Provides a view that generates an e-mail containing the data that was posted.
`infocollector`	Provides views for managing collected information.
`layout`	Provides a view that makes it possible to use alternative pagelayouts.
`notification`	Provides an interface to the notification engine.
`package`	Provides views for importing / exporting packages.
`pdf`	Provides views for configuring PDF exports.
`reference`	Provides a view for displaying documentation generated by Doxygen.
`role`	Provides views for managing roles.
`rss`	Provides views for managing RSS imports and exports.
`search`	Provides a view that displays search statistics.
`section`	Provides views for managing sections.
`setup`	Provides the web-based Setup Wizard.
`shop`	Provides views for the Webshop (basket, wish list, order list, etc.).
`trigger`	Provides a view for managing workflow triggers.
`url`	Provides views for managing the URLs stored in the database.
`user`	Provides views for logging users in / out, password changing, etc.
`workflow`	Provides views for managing workflows, workflow groups, workflow events, etc.

XML tags

The ”XML block” datatype supports the following tags / elements:

- Headings
- Bold text
- Italic text
- Unformatted text
- Lists
- Tables
- Hyperlinks
- Anchors
- Object embedding
- Custom tags

Headings

Headings / titles can be added by making use of either the h or the `header` tag. The *`level`* parameter controls the depth of the heading. It must be a number between 1 and 6. The optional *class* parameter allows the use of a desired CSS class. Usage:

```
<h [level=””] [class=””]>Example</h>
```

or

```
<header [level=””] [class=””]>Example</header>
```

By default, the specified levels are increased by one. In other words, a level 1 header in the XML block will become a level 2 header (h2) in the resulting HTML. This is because the h1 tag is reserved for the name / main title of the content object. The headings inside the XML block will thus become subheadings of the main title. This behavior can be changed by creating an override template for the *`content/datatype/view/ezxmltags/header.tpl`* template. (It cannot be controlled from within an configuration file).

Bold text

Bold text can be achieved by using one of the following tags: b, bold or strong. The optional *class* parameter allows the use of a desired CSS class. Usage:

```
<b [class=””]>Bold text.</b>
```

or

```
<bold [class=””]>Bold text.</bold>
```

or

```
<strong [class=””]>Bold text.</strong>
```

Italic text

Italic / emphasized text can be achieved by using one of the following tags: `i`, `em` or `emphasize`. The optional *`class`* parameter allows the use of a desired CSS class. Usage:

```
<i [class=""]>Emphasized text.</i>
```

or

```
<em [class=""]>Emphasized text.</em>
```

or

```
<emphasize [class=""]>Emphasized text.</emphasize>
```

Unformatted text

The `literal` tag can be used to output unformatted text, for example program source code, HTML code, XML content, etc. Everything that is inside a literal block will be rendered in the same way (character by character) as it is within the literal tags. (The text will be output using the HTML `pre` tag). The optional *`class`* parameter allows the use of a desired CSS class. Usage:

```
<literal [class=""]>Example</literal>
```

Lists

To create lists in the same way as in HTML, use the `ol`, `ul` and `li` tags. The lists can be nested. The optional *`class`* parameter allows the use of a desired CSS class. The following examples demonstrate the usage of ordered and unordered lists.

Ordered lists

```
<ol [class=""]>
    <li>Element 1</li>
    <li>Element 2</li>
    <li>Element 3</li>
</ol>
```

Unordered lists

```
<ul [class=""]>
    <li>Element 1</li>
    <li>Element 2</li>
    <li>Element 3</li>
</ul>
```

Tables

Tables can be created in the same way as in HTML using `table`, `tr`, `th` and `td` tags. The tables can be nested. Usage:

```
<table [class=""] [border=""] [width=""]>
...
</table>
```

The *class*, *border* and *width* parameters are optional. The *class* parameter can be used to assign a desired CSS class. The *border* parameter can be used to set a border (number of pixels). The *width* parameter can be used to control the table width (either 0-100% or number of pixels). Table content should be written according to normal HTML table syntax with `tr`, `th` and `td` tags as described below.

Table rows

Table rows can be created in the same way as in HTML:

```
<tr>Table row content goes here.</tr>
```

Table headers

Table headers can be created in the same way as in HTML:

```
<th [class=""] [width=""] [rowspan=""]
        [colspan=""]>Example.</th>
```

All parameters are optional. The *class* parameter can be used to set the desired CSS class. The *width* parameter can be used to set the width (either as percentage or number of pixels). The *rowspan* and *colspan* parameters are the same as in HTML.

Table cells

Table data / cells can be created in the same way as in HTML:

```
<td [class=""] [width=""] [rowspan=""]
        [colspan=""]>Example.</td>
```

All parameters are optional. The *class* parameter can be used to set the desired CSS class. The *width* parameter can be used to set the width (either as percentage or number of pixels). The *rowspan* and *colspan* parameters are the same as in HTML.

Hyperlinks

Hyperlinks can be inserted by making use of the `a` or the `link` tags. Usage:

```
<a href="" [target=""] [ class=""] [title=""]
        [id=""]>Example.</a>
```

or

```
<link href="" [target=""] [ class=""] [title=""]
        [id=""]>Example.</link>
```

The *href* parameter is required and it must be set to a valid address (either external or internal). The *target* parameter can be used to determine how the target URL should be opened (inside the existing / active browser window / tab or within a new window / tab). The *class*

parameter can be used to specify a CSS class that should be used when the link is rendered. The *title* parameter can be used to specify a short description that will be shown when the mouse pointer is hovering over the link. The id parameter is for assigning unique identifiers.

Internal links

It is possible to create internal links (to other nodes and objects) by making use of the `eznode://` and the `ezobject://` notation. The internal links will be created dynamically based on the node / object ID numbers. In other words, if a node is moved, the link(s) will point to the new location(s) and thus they will not be broken.

Link to a node

A link to a node can be created either by specifying the target node's ID number or the node path. The following examples demonstrate an internal link to a node numbered 128.

```
<a href="eznode://128">Example.</a>
```

or

```
<link href="eznode://128">Example.</link>
```

The following examples demonstrate how an internal link to a node located at "products/computers/example" can be created.

```
<a href="eznode://products/computers/example">
        Example.</a>
```

or

<link href="eznode://products/computers/example">
Example.</link>

Link to an object

The following examples demonstrate how an internal link to object number 1024 can be created.

```
<a href="ezobject://1024">Example.</a>
```

or

```
<link href="ezobject://1024">Example.</link>
```

When object linking is used, the destination address will be generated using the main node assignment of the target object.

Anchors

The "anchor" tag makes it possible to insert HTML anchors inside the XML block. The inserted anchors will work like standard HTML anchors. Usage:

```
<anchor name="" />
```

The *name* parameter must be set to a unique identifier for the anchor. Anchors can be reached by appending the hash character (#) followed directly by the name of the anchor that the browser should jump to, for example "http://www.example.com/hobbies#music").

Object embedding

The embed tag makes it possible to insert an arbitrary content object directly in the XML block. It can for example be used to embed images. Usage:

```
<embed href="" [view=""] [align=""] [target=""]
        [size=""] [id=""] />
```

Parameter	Description	Required
href	The *href* parameter must be a valid link to either a node or an object using the same notation as for hyperlinks (for example "eznode://134", "eznode://path/to/some/node" and "ezobject://1024"). If the provided link is a link to a node, eZ publish will use the object that is encapsulated by that node. In other words, in both cases it is the object that will be inserted (the node notation is just a wrapper).	Yes
view	The *view* parameter makes it possible to specify the view that should be used when the object is rendered (for example "full", "line", etc.).	No
align	The *align* parameter can be used to specify the positioning of the embedded object. Possible values are left, center and right.	No
target	The *target* parameter can be used to set the opening method (same browser tab / window or new browser tab / window) for the embedded item (for example _self, _blank, etc.).	No
size	The *size* parameter can be used to set the image size that should be used when an image object is included (for example "small", "medium", "large", etc.).	No
id	The *id* parameter makes it possible to assign a unique ID which will be the ID attribute in the resulting HTML.	No

Custom tags

In addition to the default tags described above, the "XML block" datatype makes it possible to use custom tags. A custom tag can be used both as a block or an inline element. Custom tags must be specified using the AvailableCustomTags[] array in the [CustomTagSettings] block within an override for the *content.ini* configuration file. (This is also where you can specify whether the tag should be a block or an inline element.) When the XML is rendered, the contents of a custom tag will be replaced by a custom template. The name of the template must be specified using the *name* parameter. Example of usage:

```
<custom name="template_name"
        [custom_parameter="value" [...] ]>
The quick brown fox jumps over the lazy dog.
</custom>
```

The custom tag in the example above will be replaced by a template called *template_name.tpl*. This template must be located in the following directory within the current design: *templates/content/datatype/view/ezxmltags/* (or one of the fallback designs). It is also possible to create an override template. The contents of the tag will be available in the `$content` variable within the inserted template. The custom parameters are optional. When used, a custom parameter will be available as a template variable with the same name as was specified in the tag itself.

Glossary

CSS	*Cascading Style Sheets* - a construct that works with HTML to give both web developers and user agents more control over page display. CSS allows designers to create style sheets that define how different elements (for example headers, links, images, etc.) should appear. The style sheets can then be applied to any web page. Changing the stylesheet will result in a change of all web pages using that style sheet.
DNS	*Domain Name System* (or Service or Server) - an internetbased service that translates domain names (such as `www.example.com`) into IP addresses (for example 60.70.12.130) and vice versa. While humans prefer to work with domain names, computers work with IP addresses; the DNS provides an interface.
HTML	*Hyper Text Markup Language* - a language designed for the creation of web pages and other information viewable in a browser. HTML is typically used to structure information, such as headings, paragraphs, lists, images and so on. It can be used to define the semantics of a document.
HTTP	*Hyper Text Transfer Protocol* - the primary method used to convey information on the World Wide Web.
LDAP	*Lightweight Directory Access Protocol* - a relatively simple protocol for updating and searching directories running over TCP/IP.
MTA	*Mail Transfer Agent* - an application that handles sending and receiving e-mails to and from the host / system where it is installed.
SOAP	An XML-based lightweight protocol for exchanging information.
SMTP	*Simple Mail Transfer Protocol* - a protocol used to send email messages between computers.
PDF	*Portable Document Format* - Adobe Systems' widely used file format that allows the electronic representation of documents in a paged form. It is supported on all major computer platforms.

PHP	*PHP Hypertext Preprocessor* - an open source, serverside HTML-centric scripting language that can be used to create dynamic web pages.
Unicode	A 16-bit character encoding scheme that supports all languages (Western, Eastern, Cyrillic, Greek, Arabic, Hebrew, Chinese, Japanese, Korean, Thai, Urdu, Hindi, etc.) within a single character set. In addition, the specification includes standard compression schemes and a wide range of typesetting information required for worldwide locale support.
URL	*Uniform Resource Locator* - a way of specifying the location of something on the internet, for example `http://www.example.com/hello.html`. The part before the colon specifies the protocol; the part after the colon specifies a unique address.
UTF-8	An encoding scheme for storing Unicode code points in terms of 8-bit bytes. Characters are encoded using sequences of 1, 2, 3 or 4 bytes. Characters in the ASCII character set are all represented using a single byte.
WebDAV	*Web-based Distributed Authoring and Versioning* - a set of extensions to the HTTP protocol that allow users to collaboratively view, edit and manage files on remote web servers.
XHTML	*eXtensible Hypertext Markup Language* - a reformulation of HTML 4.0 in XML 1.0. XHTML is a new and widely used language for creating web pages.
XML	*Extensible Markup Language* - a rich and highly portable format for defining complex documents and data structures.

Index

Symbole

A

B

C

D

N

O

P

R

S

T

U

V

W