



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

concrete5 for Developers

Learn to develop themes, blocks, and attributes with concrete5

Sufyan bin Uzayr

[PACKT] open source*
PUBLISHING community experience distilled

concrete5 for Developers

Learn to develop themes, blocks, and attributes
with concrete5

Sufyan bin Uzayr



BIRMINGHAM - MUMBAI

concrete5 for Developers

Copyright © 2014 Packt Publishing

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

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

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

First published: October 2014

Production reference: 1241014

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-355-2

www.packtpub.com

Credits

Author

Sufyan bin Uzayr

Project Coordinator

Kranti Berde

Reviewers

Chris van Dam

Michele Locati

Sarunas Narkevicius

Proofreaders

Simran Bhogal

Maria Gould

Ameesha Green

Commissioning Editor

Mary Jasmine Nadar

Indexers

Hemangini Bari

Mariammal Chettiyar

Acquisition Editor

Llewellyn Rozario

Production Coordinator

Kyle Albuquerque

Content Development Editor

Ruchita Bhansali

Cover Work

Kyle Albuquerque

Technical Editor

Mrunmayee Patil

Copy Editors

Rashmi Sawant

Stuti Srivastava

About the Author

Sufyan bin Uzayr has experience and an interest in everything related to web design and development. He has worked with numerous content management systems so far and writes about web design, web development, and typography for several blogs and magazines of repute. He also has a background in Linux administration, database management, cloud computing, and web hosting.

Sufyan is an open source enthusiast. He can code in PHP, RoR, Perl, and is also proficient in JavaScript, jQuery, HTML5/CSS3, and several other web development trends.

Apart from technology and coding, he also takes a keen interest in topics such as history and politics, and is the author of *Sufism: A Brief History*, Notion Press. Furthermore, he also serves as the editor-in-chief of various progressive journals and news publications, including *The Globe Monitor*.

Sufyan blogs at <http://www.codecarbon.com>; you can learn more about his writings and works at <http://sufyanism.com/>.

Acknowledgments

This book owes its existence to the efforts of several people who helped me out at various stages of the writing process. I would like to thank:

- The creators of concrete5 – this is a wonderful CMS and I'm sure the readers of this book will agree with my statement.
- The concrete5 documentation and third-party hobby blogs because this is where I learned how to use concrete5, many years ago! Some of those third-party blogs might still be alive, though most of them are either dead or defunct by now.
- The editors, reviewers, and proofreaders for ensuring that the book's quality is top-notch and all the bases are covered.
- Packt Publishing – Mohammad Rizvi for all his help before, during, and after the book preparation process; Ruchita Bhansali for being super patient with my rather questionable adherence to the deadlines as well as for all her help with the formatting of the drafts, and for being available all day long via Skype, each time I had a query; and also Mrunmayee Patil, for her help with technical edits and preparation of final drafts and layout of the book, and for working tirelessly even on a holiday, just to ensure that this book came out in an impeccable shape.
- Everyone else who is closely or remotely involved with this book – layout and cover designers, promoters, and well-wishers.
- Loved ones – family, teachers, mentors, and friends – for being there!

About the Reviewers

Michele Locati has an interest in computer programming that dates back to the late 80s when he was just a young boy and started programming in BASIC for his glorious Amstrad CPC 6128.

Then he moved to programming in C and assembly when he had his hands on an Amiga, and then he switched to C++ for his initial programs for MS-DOS.

Michele's interest in creating websites started in the late 90s, when he saw the Internet for the first time.

Because of his love for programming, after graduating in electronic engineering (in the biomedical field) from the Politecnico di Milano University, in 2000, he started working in a start-up company located in Bergamo (northern Italy) that creates PC applications and web services based on the customers' requirements.

In 2010, Michele moved to Como—a wonderful place, because of his wife, and started working for a company that produces CAD software, managing the company websites and intranets.

Since 2012, he has been participating in the concrete5 open source project, being one of the main community contributors. He is mainly focused on localization issues and integration with the Transifex platform, but greatly contributed in all the aspects of the project, from the core PHP features to the auxiliary Node.js and Grunt tasks.

Sarunas Narkevicius has been working with concrete5 and Magento development for more than 5 years. He has reviewed *concrete5 Beginner's Guide, Second Edition*, Packt Publishing.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Building a concrete5 Theme	5
Themes and page types	5
Getting started with theme building	6
The development tools and environment	7
Designing templates – things you should know	8
Types of websites	8
Navigation menus	9
Images	9
Other design issues	10
Creating the theme	10
Installing the template	10
The theme's thumbnail	11
The theme's description	11
The theme's style sheet	12
Creating the default page type	13
The <head> tag	13
The <body> tag	14
Content areas	14
The main or primary content	15
Sidebar	16
Navigation	17
Footer	18
Miscellaneous	18
A single page template	19
Packaging your theme for the marketplace	19
Submitting your theme to the marketplace	20
Summary	20

Chapter 2: Introduction to concrete5 Blocks	21
The directory structure of concrete5 blocks	22
The database – db.xml	22
The controller – controller.php	22
The icon – icon.png	22
The core templates	23
Custom view templates	23
Assets	23
Additional tools	24
The MVC framework in concrete5	25
Creating a new block	27
Validating user inputs	31
Summary	34
Chapter 3: Advanced concrete5 Blocks	35
Using database tables	35
Database connections	36
Database queries	37
Getting data from the database	38
The GetOne command	38
The GetRow command	39
The GetAll command	39
The database debugging mode	41
Working with advanced concrete5 blocks	41
The composer	41
Working with file management	42
Overriding the default functionality of concrete5	42
How to override?	43
Packaging your code for distribution	44
Packages in concrete5	44
The package controller	45
Summary	46
Chapter 4: Attributes in concrete5	47
An overview of the concrete5 attributes	47
Attribute terminology in concrete5	48
Understanding and working with attribute terminology in concrete5	49
The attribute category	49
Creating an attribute category	50

The attribute key	50
Creating a database file for the attribute keys	51
Creating a class file	51
Creating a view page for the attribute category	55
The attribute type	57
The database file	57
The attribute type controller	58
Creating a custom attribute type in concrete5	60
The database file	62
The form.php file	62
The controller.php file	63
The PacktAddressAttributeTypeValue class	68
Installation of the custom attribute type	69
Using the new custom attribute type	70
Summary	70
Chapter 5: Permissions and Workflows	71
Introducing permissions in concrete5	72
The difference between simple and advanced permissions	72
Simple permissions	73
Advanced permissions	74
A closer look at advanced permissions in concrete5	75
Enabling advanced permissions	75
Assigning permissions	75
Inclusion and exclusion	77
Timed permissions	78
Copying permissions	80
A note on area and block permissions	81
Managing advanced permissions programmatically in concrete5	82
Workflows in concrete5	84
More about the basic workflow	85
The workflow list	85
The Waiting for Me section	86
Setting up the basic workflow	86
Creating the workflow	86
Attaching workflow to page permissions	90
Modus operandi	92
Summary	93
Index	95

Preface

This book introduces you to web development with concrete5. You will learn how to use concrete5 to create and extend websites. First, you will start with theme development, and then work with concrete5 blocks. Apart from using basic blocks, you will also be working with advanced blocks, and then move on to database operations, attributes, and attribute types. Lastly, you will focus on permissions and workflows in concrete5. This book will serve as a companion to help you master concrete5 with ease.

What this book covers

Chapter 1, Building a concrete5 Theme, talks about building a concrete5 theme from scratch as well as extending an existing HTML template to create a concrete5 theme.

Chapter 2, Introduction to concrete5 Blocks, introduces you to concrete5 blocks and the underlying MVC structure that concrete5 is well-known for. It also talks about blocks and their role in overall concrete5 development as well as the creation of concrete5 blocks, validation of user inputs, and other related topics.

Chapter 3, Advanced concrete5 Blocks, discusses advanced concrete5 blocks and add-ons. The chapter also talks about database operations, especially getting data from the database, database debugging, and database connections along with file management in concrete5.

Chapter 4, Attributes in concrete5, focuses on attributes and attribute types. The chapter also deals with creating custom attribute types.

Chapter 5, Permissions and Workflows, looks at how to manage permissions in concrete5, simple and advanced permissions, programmatically dealing with permissions, and then we move on to learn about workflows of concrete5 and their configuration.

What you need for this book

Working knowledge of basic web development is required. You will need to be aware of CSS/HTML in order to properly tweak concrete5 themes. Plus, a background in PHP is needed for advanced operations with code. Database skills are not required, but can come handy.

In terms of software, concrete5 can be run either on a LAMP or WAMP stack with ease. You will need a Linux or Windows server with Apache or LiteSpeed (the latter being proprietary) installed as well as MySQL with PHP. In this book, I have used a Linux server running Apache.

Who this book is for

Whether you have had some previous experience with concrete5 or are entirely new to it, this book will help you understand all that you need to know in order to get started with concrete5 development. A background in PHP is required; some knowledge of HTML/CSS is needed in order to fully grasp the concepts underlying concrete5 theme development. Knowledge of databases, though not entirely necessary, could come in handy.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You simply append the following line to your `site.php` file located in the `/config/` folder."

A block of code is set as follows:

```
define('PERMISSIONS_MODEL', 'advanced');
```

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

```

```

New **terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Navigate to **Developers** | **Submit to Marketplace** on the concrete5 website."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

Errata

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

Piracy

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

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

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

Questions

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

1

Building a concrete5 Theme

This chapter talks about building a concrete5 theme from scratch. In concrete5, themes are actual directories within the filesystem that contain the code and design elements required to properly display the pages. Thus, every page that you create in concrete5 needs to have a theme assigned to it.

Here is a list of everything that we will be discussing in this chapter:

- The theme structure in concrete5
- Creating a concrete5 theme from a static HTML template
- Things you should know when packaging and publishing a theme to the marketplace
- An overview of the theme structure in concrete5

Themes in concrete5 contain several template files, which include HTML, CSS, and PHP code that defines editable regions on the given pages and other details related to your website. If you have had any experience working with themes for other CMSs such as WordPress, you will easily be able to find your way through the code in concrete5, because both the CMSs employ PHP. However, concrete5 allows for greater flexibility and customization compared to WordPress, as themes are basically dependent on pages. To help you understand this better, I will first start by explaining the manner in which themes and page types work in concrete5.

Themes and page types

Each time a page is loaded, concrete5 tries to determine the theme for that particular page. Thereafter, on the basis of the concerned page type, the relevant template file is loaded. For example, if the page you are loading is a single page, the `view.php` file will be used. If, however, a template file that matches the page type handle is not found, the `default.php` file is used.

Basically, every page in concrete5 has a page type. Thus, your homepage is the default page type; your `view.php` file is the single page type, and so on. Your theme needs to have certain page layouts and types, apart from design elements such as images and style sheets. Your users can modify the pages by adding blocks to editable regions, such as the sidebar, header, main content, and so on.

Therefore, usually, a general concrete5 theme has the following editable areas:

- Header Nav (which holds the auto-nav block and is used for navigation menus)
- Main (the main section of the given page)
- Sidebar

There can also be areas for headers and footers, if required.

So, what should your theme have? Basically, apart from style sheets and images, you need to build page templates. In general, a concrete5 theme has the following page templates by default:

- `default.php`: This is the home page template. It is required.
- `view.php`: This is the single page template. It is always a good practice to have this file.
- `left_sidebar.php`: This is the left-sidebar template. It is not required; add it if you wish to.
- `full.php`: This is a page without the sidebar and is not always required.

You just need to properly code the `default.php` template. The `view.php` template can be built by duplicating `default.php` and adding certain lines of code, as we shall see later in this chapter. Similarly, `full.php` and `left_sidebar.php` are mere duplicated versions of the `default.php` template with some additional CSS styling.

As we have already mentioned, the template files for concrete5 themes are coded using HTML, CSS, and PHP. As a developer, you might find it convenient to start with static HTML templates and thereafter, convert them to concrete5 themes.

Getting started with theme building

Now that we have understood the theme structure and how page types work in concrete5, let's start with the creation of our first concrete5 theme.

There are many ways in which you can get started with theme development in concrete5. First, you can simply customize a template for your website and tweak it to suit your needs. However, if you are serious about theme development and wish to create a proper, full-fledged standalone theme for concrete5, there are certain things that you need to bear in mind, and we will discuss them here.

Since we are talking about concrete5 themes, I am assuming that you either have the design layout ready, or you would be able to take care of this. If not, you can follow the popular route and get a static HTML template and convert it to a theme that works with concrete5.

Before you actually go template-hunting and start building your first concrete5 theme, you need to be sure that you have the development environment ready and at your service. After all, you cannot code a theme unless you have the coding tools at hand, can you?

The development tools and environment

There are not many things or tools that you require in order to work with the concrete5 development. Here is what you should have on your system or server, depending on where you choose to perform the development (assuming that you have already worked with the PHP development at some point of time, you might already have all of this anyway):

1. For the development environment, you need to set up either WAMP (<http://www.wampserver.com/en/>) or MAMP (<http://www.mamp.info/en/index.html>), depending on whether you're using Windows or Mac OS. If you're a Linux user like me, LAMP (<http://www.lamphowto.com/>) is the way to go! Vagrant is also a good option that you can consider using.
2. Next, download and install concrete5. Be sure to check your theme on a fresh installation so as to avoid conflicts across themes and extensions. More importantly, be sure to create some sample content after installing concrete5—add some blocks and forms to the content area as well as the sidebar, and so on, just to check how your theme looks with the most common elements. You should note that the concrete5 setup already comes with options that help you add some content during the installation phase.
3. That's all! You can, of course, use specialized tools such as Firebug if that's how you prefer working on your web development projects.

Once you have the development prerequisites set up, you are good to go. You can now start with the theme development, and in order to do that, you need to grab a website template. However, not all templates are coded alike, so you need to pick the best one for your concrete5 theme.

Designing templates – things you should know

Often, many concrete5 developers end up picking the wrong static templates for conversion to a concrete5 theme. Much like any other CMS, concrete5 themes too have certain characteristic features of their own, and as a result, you need to be aware of the limitations and prerequisites of static HTML templates before deciding which one you should be converting to your concrete5 theme.

Bluntly put, the static template that you are working with should be flexible enough to work as a concrete5 theme. The beauty of concrete5 lies in its flexibility. You can add content to virtually any type of page using the block system, and this is where your theme should excel. It should support the addition of blocks in the best manner possible, because if your theme limits the user's ability to add content via blocks, it has defeated the very purpose of concrete5. Obviously, not all static HTML templates can work with such flexibility, so you need to be sure that the one you're choosing has a flexible layout that can allow for easy addition of content.

While there are no rigid rules when it comes to choosing static template layouts, I have tried to put together some basic pointers that will help you pick the right template for your concrete5 theme.

Types of websites

Basically, there are two major types of websites that you need to distinguish between: blogging sites and all other sites. The reason for this distinction is that blogging sites are updated frequently and tend to show the latest posts on the home page, so their static templates too have a different layout as compared to other generic websites. In general, most blogging templates tend to have a more or less similar layout: a main content area for blog posts with information, such as excerpts, the blog author's name, date and time stamps, the comment count, and so on, adjacent to a sidebar that contains stuff such as post categories, tags, search bars, and other relevant widgets.

concrete5, however, is not entirely a bloggers' CMS. While it can surely be used for blogs, its primary audience does not include bloggers and is, therefore, more of a mainstream CMS that is meant for general-purpose websites.

So, can we not use a blogging template to build a concrete5 theme? Actually, yes we can! However, it is worth noting that certain blog-only features might not be supported by concrete5 themes, for instance, design elements for a blog's tag clouds, and so on. Similarly, the search bar, which is a common feature of HTML templates, will not be functional in a concrete5 theme (this should not be a concern though, because concrete5—just like every other CMS—offers a search functionality, but you should know this bit when converting a static template into a concrete5 theme). You can, however, always develop your own block type or use a package from the marketplace. Also, if all else fails, you can just hardcode a search box and include a single `/search` page.

Navigation menus

When it comes to your website's navigation menus, you need to focus on two key aspects. First, concrete5 themes should, ideally, have text-only navigation menus so as to be flexible enough to handle different pages and page types. Certain static templates tend to have images in navigation menus, and you should avoid such templates. Alternatively, you can add a new attribute to the pages that describe which one is the image, and then use the attribute value to build the menu.

Secondly, concrete5 has an auto-nav block that outputs an unordered list for the navigation menus. What this means is that if you want your template to work out of the box as a concrete5 theme, you should look for templates that are coded with unordered lists for navigation menus (look for `` or ``). However, auto-nav can also be themed to not use `` or `` tags without much effort.

Images

Any template or theme generally has two types of images: background images and primary images.

For background images, flexibility becomes the key. Often, certain HTML templates have background images that work perfectly well in a static layout but are not flexible or symmetrical enough to be used with different layouts (remember the beauty of concrete5 lies in the fact that it is flexible enough to allow multiple page types and layouts). For example, a background image with two lines running through it might work for a three-column layout but not for a two-column or single-column layout. Since background images are called via CSS and are not as easily editable as text, your users will have no way of modifying them without using image editors and uploading the newer images. To avoid hassles for your users, be sure to check that the background image of the template in question is flexible enough to permit usage across multiple page types.

For primary images, just look for the `` tags. Say, for header images and banners, if the template has embedded them via CSS (much like background images), the template will, again, not be flexible enough to be used for concrete5. This is because the primary images will not be changeable using concrete5's block system. You can also do this using page attributes or via the global dashboard configuration page. However, if the primary images are implemented as actual `` tags in the HTML code, you can make them editable regions in your concrete5 theme, thereby allowing for custom attributes and more flexibility.

Other design issues

Apart from template layouts, images, and navigation menus, there are few other design metrics that you should bear in mind when looking for a static template that can be converted to a concrete5 theme.

Be sure that regions such as the site's title area have enough space for longer pieces of text. Furthermore, check for the manner in which boxes are styled in the template. For example, certain templates use very specific CSS rules to style their boxes through the page and leave little room for flexibility. Owing to such specific CSS rules, some blocks in your resultant concrete5 theme might probably never be called. To avoid this, opt for templates that are not rigidly coded using specific CSS rules. Alternatively, you can also add some CSS rules to a `<div>` tag that contains the area.

Once you have figured out which template is to be used as the base for your upcoming theme, you can start coding!

Creating the theme

We are now ready to start building our theme. We will first start by installing the template, followed by working on the theme's style sheet and default page types.

Installing the template

Now, it is time to get started with the process of theme creation. Let's first install the template:

1. After downloading and unzipping said template, upload it to the `/themes` directory of your development site. Be sure to place all the template files inside a subfolder of their own if they haven't been placed already.
2. Check whether all the images that are utilized by the template, or the templates that you are planning to use in your theme, are in a subfolder named `images` within the theme's folder.

3. For example, the path to your theme and its images should now look like `my-site-root/themes/theme_name/images`.
4. If your theme's name has spaces in it (nearly all web template names have spaces), now is a good time to replace these spaces with underscores. The name of the theme's folder should be in lowercase.
5. You can also consider renaming the `index.htm` file to `default.php` so that it works as the default page template and concrete5 treats it as a page type. You will, however, need to clean up the `index.htm` file in order to make it work as the default page type. More on how to do this is discussed later in this chapter.

All done! You have uploaded the template and are one step closer to your concrete5 theme.

The theme's thumbnail

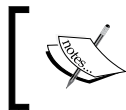
You will need to create a thumbnail image for your theme. You can easily do this by grabbing a preview of your static template and resizing it to 120 px by 90 px. Once done, save it as a PNG file (name it `thumbnail.png`) and upload it to your theme's folder.

This thumbnail will be used to show your theme in the **PAGES & THEMES** section of the concrete5 **Dashboard**.

The theme's description

Now, create a description file for your theme. Simply create a text file, write your theme's name in the first line and its description or whatever information you deem appropriate in the second line, and save it as `description.txt`.

This file belongs in the `theme` folder too and will be used to show the theme description in the **PAGES & THEMES** section of the concrete5 **Dashboard**.



Now the actual coding begins. I will be describing the coding process step by step and giving code samples wherever necessary. I am assuming that you have a working knowledge of PHP, HTML, and CSS.

The theme's style sheet

We will now take a look at the theme's style sheet and edit it accordingly:

1. For the sake of convenience, you might want to rename the theme's style sheet to something easy to remember, say, `main.css`.
2. Also, be sure that the image reference in the style sheets should not be surrounded by code.
3. In concrete5, the **WYSIWYG** editor (TinyMCE, by the way) looks specifically for a typography style sheet in order to display and format the text in the editor box in the same way as it is going to appear on the page. So, you will need to create a separate style sheet named `typography.css`.
4. Now, we need to populate our `typography.css` style sheet from `main.css`. Basically, the idea is to search for and move certain typography-related properties. Search for the following elements in your `main.css` style sheet:
 - `body`
 - `aa:`
 - `hover`
 - `ul`
 - `li`
 - `ol`
 - `p`
 - `address`
 - `pre`
 - `h1`
 - `h2`
 - `h3`
 - `h4`
 - `h5`
 - `h6`
 - `div`
 - `block`
 - `quote`
 - `cite`

5. Move the following properties to `typography.css` if they appear anywhere in the preceding elements:
- `color`
 - `font`
 - `font-family`
 - `font-style`
 - `font-variant`
 - `font-weight`
 - `font-size`
 - `line-height`
 - `word-spacing`
 - `letter-spacing`
 - `text-decoration`
 - `text-transform`
 - `text-align`
 - `text-indent`
 - `text-shadow`

All the other elements and properties remain in `main.css`, since they are not related to typography.

Creating the default page type

Now, on to our `default.php` file. We will need to work on the `<head>`, `<body>`, and other sections.

The `<head>` tag

It is customary to include the head and footer in the `/elements/header.php` and `/elements/footer.php` files, respectively.

Replace everything inside `<head>` with the following lines:

```
<head>
    <?php Loader::element('header_required'); ?>
    <link rel="stylesheet" type="text/css" href="
        <?php print $this->getStyleSheet('main.css'); ?>" />
```

```
<link rel="stylesheet" type="text/css" href="
    <?php print $this->getStyleSheet('typography.css'); ?>" />
</head>
```



Downloading the example code

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

Eliminate all the meta info and title details that the template might otherwise have. The `<?php Loader::element('header_required'); ?>` tag will take care of `<title>`.

The `<body>` tag

If your template shows the site name or title in the `<body>` section, replace it with this:

```
<a href="<?php echo DIR_REL?>/">
<?php $block=Block::getByName('Site_Name');

</a>
```

Content areas

Now, we need to build the editable content areas of our theme. You can have whatever content areas you want on a page, but by default, concrete page types have these content areas: main, sidebar, header navigation, and sometimes footer.

If you intend to publish your theme in the concrete5 marketplace, I would advise you to have the content areas on your pages, because these are the standard editable regions that almost everyone prefers. As a theme-maker, you just need to provide content regions to your users and let them put the appropriate blocks wherever they wish to.

The main or primary content

In your template, locate the HTML content. It should look something like this:

```
<div id="content">
  <div class="post">
    <h2 class="title"><a href="#">Title </a></h2>
    <p class="meta"><span class="date">Date</span></p>
    <div style="clear: both;">&nbsp;</div>
    <div class="entry">
      <p> some text</p>
      <p>more text</p>
    </div>
  </div>
  <div class="post">
    <h2 class="title"><a href="#">Title</a></h2>
    <div style="clear: both;">&nbsp;</div>
    <div class="entry">
      <p>some text</p>
    </div>
  </div>
  <div style="clear: both;">&nbsp;</div>
</div>
```

Replace the HTML with this code:

```
<div id="content">
  <?php
    $a = new Area('Main');
    $a->display($c);    ?>
  <div style="clear: both;">&nbsp;</div>
</div>
```

This will take out all of the design content but still retain the structure of the page.

Sidebar

Coding the sidebar is pretty simple. This is based on the assumption that this is what your template's sidebar looks like:

```
<div id="sidebar">
  <ul>
    <li>
      <div style="clear: both;">&nbsp;</div>
    </li>
    <li>
      <h2>Categories</h2>
      <ul>
        <li>Item A</li>
        <li>Item B</li>
      </ul>
    </li>
  </ul>
</div>
```

Replace the HTML code with this:

```
<div id="sidebar">
  <ul>
    <?php
      $a = new Area('Sidebar');
      $a->setBlockWrapperStart('<li>');
      $a->setBlockWrapperEnd('</li>');
      $a->display($c);
    ?>
  </ul>
</div>
```

However, if your sidebar does not use unordered lists and has a series of `<div>` elements instead, use this code:

```
<div id="sidebar">
    <?php
        $a = new Area('Sidebar');
        $a->setBlockWrapperStart('<div>');
        $a->setBlockWrapperEnd('</div>');
        $a->display($c);
    ?>
</div>
```

What are we doing in the preceding code? Nothing much! We are just telling concrete5 to wrap each block's content within the concerned element, which can be either an unordered list or a `<div>` element. This block wrapper will ensure that the sidebar is displayed properly.

Navigation

Global Area is the section that is generally used for navigation purposes in concrete5.

Remember the part about your HTML template with a text-only navigation menu, preferably with unordered lists? This is where it comes handy! If your template has a navigation menu with unordered lists, just replace it with this:

```
<div id="menu">
    <?php
        $a = new Area('Header Nav');
        $a->setBlockLimit(1);
        $a->display($c);
    ?>
</div>
```

What does this do? Each time an auto-nav block is placed by the user, the nav menu will be ready to go. The `setBlockLimit(1)` parameter will ensure that the user does not add more than one block here, so any confusion regarding the sidebar is prevented.

Footer

There is not much to talk about the footer. However, you might want to make life easier for your users by allowing them to easily integrate code for Google Analytics, and so on. For this, you need to place code that outputs the tracking code JS for this purpose. Here it is:

```
<?php Loader::element('footer_required'); ?>
```

Place the preceding code just above the `</body>` tag.

The `footer_required` element also performs certain other functions, such as adding the toolbar for editors, and so on.

Miscellaneous

You can either choose to hardcode images in your theme, or make them editable content regions in their own right. To hardcode images, replace their `src` attribute with a concrete5 path function as follows:

```

```

Or, to make them editable content areas, replace the `` elements with this:

```
<?php
$a = new Area('My name');
$a->display($c);
?>
```

If you notice that the concrete5 toolbar is interfering with your theme's layout, it is probably because your theme's style sheet and concrete5's admin style sheet are both trying to do the same job. If so, you need to use a `div` wrapper. Open the page template, and simply add `<div class="c5wrapper">` just after `<body>` and add `</div>` just before `</body>`. Next, in the `main.css` file, change the `{ body` to `.c5wrapper {`.

A single page template

Creating a single page template is easy. Just duplicate your `default.php` file, rename it `view.php`, and then open and locate the following code:

```
<?php
    $a = new Area('Main');
    $a->display($c);
?>
```

Replace it with this:

```
<?php
print $innerContent;
?>
```

Done!

Similarly, if you wish to create additional page types, just copy the `default.php` file, rename it accordingly, and then make the necessary CSS tweaks.

Packaging your theme for the marketplace

We are now ready to package our theme for the concrete5 marketplace. Much like any other marketplace, the concrete5 marketplace has its own set of rules and guidelines, which are mentioned as follows:

- **Browser testing:** The concrete5 marketplace demands that you test your theme across multiple browsers. Be sure that your theme is compatible with various browsers before applying for submission.
- **License:** Your theme needs to have a license. You can opt for MIT, GPL, or use the default concrete5 marketplace license, which is mostly meant for premium themes.
- **Icon:** Place a PNG file named `icon.png` in the root folder of your theme. Its dimensions should be 97 px by 97 px, and it should have 4 px rounded corners. This will serve as the default icon of your theme in the marketplace.
- **Changelog:** Create a changelog file for your theme, telling the world what has changed per version update. These files will need to be placed in your theme's main folder and not in a subfolder.

Submitting your theme to the marketplace

Navigate to **Developers | Submit to Marketplace** on the concrete5 website. You will need to log in, and then you can upload your theme. If you are offering screenshots, keep them 400 px by 400 px. Also, if it's a premium theme that you are selling, concrete5 will retain 40 percent of the sale's price.

Summary

This brings us to the end of this chapter. We have mastered theme creation in concrete5. We learned how to pick the perfect template, and then we learned how to extend it to create a concrete5 theme. You should now understand how to prepare a concrete5 theme, package it for the marketplace, and/or use it in your websites.

In the next chapter, we will get started with block development in concrete5. First, we will focus on simple block development in order to better understand the concept of blocks in concrete5 and how to use them to the fullest. Thereafter, we will progress to advanced concrete5 blocks. The next few chapters will also involve some database operations, so we will also be targeting the concrete5 database and its mode of functioning.

2

Introduction to concrete5 Blocks

This chapter will introduce you to concrete5 blocks and the underlying model-view-controller structure that concrete5 is well known for. Basically, we will be talking about blocks and their role in the overall development of concrete5 as well as the creation of concrete5 blocks, validation of user inputs, and other related topics.

Here is what we will be discussing at length in this chapter:

- The directory structure of concrete5 blocks
- The model-view-controller framework and the controller class
- Building a basic block
- Validating the user input inside a block

We will, however, discuss the advanced techniques used to build concrete5 blocks and add-ons in the next chapter.

We will now start with concrete5 blocks. First, we need to understand what blocks are, how they are organized, and their directory structure.

In concrete5, a block represents the smallest presentational unit. It can be added to a page, and then you can store values or data using it. Thus, a block is basically an entity that is added to a concrete5 page and then used to store the visible data.

As soon as you add a block to a given page, a block object is created, which in turn can be copied or aliased across different pages as per your needs. Also, since the blocks are primarily used to hold content, each block has a `BlockType` object associated with it, which tells concrete5 what type of data the said block can store, such as the slideshow, page list, and so on.

The directory structure of concrete5 blocks

Before we move ahead with the actual development and creation of concrete5 blocks, let's spend a few moments familiarizing ourselves with the contents of the average block's directory. This will help us gain a better understanding of the importance and usage of individual files of concrete5 blocks.

The database – db.xml

Every concrete5 block has a file named `db.xml`, which takes care of the database definitions for the concerned block. Basically, the `db.xml` file does nothing except describe the schema of the block.

The `db.xml` file can contain multiple tables within itself, and its schema can be altered if required. Note that the core block's type table contained within this file must contain a column named `BID`, which is an unsigned integer.

For the technically inclined and curious-minded folks, the `db.xml` file adheres to the **ADODB AXMLS** format.

The controller – controller.php

The `controller.php` file will be used to extend the `BlockController` object and also implement the MVC approach in the process of block development. We will analyze its role and contents when discussing the actual block creation later in this chapter, and we will talk about the controller class, its variables, and methods in the *The MVC framework in concrete5* section of this chapter.

The icon – icon.png

Your block needs to have an icon, which will be displayed in the **Add Block** window. The icon is a PNG file, which is 16 x 16 pixels in dimensions and is named `icon.png`.

The core templates

Blocks in concrete5 tend to have certain core templates, which are described as follows:

- `add.php`: This file is rendered when the block is added
- `edit.php`: This file is rendered when the block is edited
- `view.php`: This file is rendered when the block is viewed

Beyond this, there is an optional `scrapbook.php` file that is rendered when the block is displayed within the scrapbook, though of late, IIRC scrapbooks are being deprecated.

Custom view templates

Complex blocks often have custom view templates of their own, which are found in the `/templates` subdirectory in the block's main directory. Such templates are available in the **Set Custom Template** dialog box. Most of the time, templates that deal with optional but enhanced features such as `breadcrumb.php` and `header_menu.php` come under **Custom View Templates**.

Much like the main `view.php` templates, custom templates too can specify their own CSS and JavaScript. You should note that there are optional alternatives to the default `view.php` file that can present the same data in a different style.

Assets

The `auto.js` file is automatically loaded as soon as the block is placed in the add or edit mode.

Furthermore, files such as `view.css` and `view.js` are automatically loaded if the block is added to a page. The files within the subdirectories named `css` and `js` follow the same model as `view.css` and `view.js`.

Additional tools

A block's directory can also contain auxiliary and less-often used files, which are commonly dubbed as **tools**. In concrete5 blocks' terminology, a tool is just a script that is executed for a given purpose. The loading of AJAX scripts and the handling of RSS feeds are noteworthy examples of tools.

To explain at length, `rss.php` is a file that exists in the `/tools` directory in the block's main directory. This is how the view template handles the display:

```
$rssUrl = $controller->getRssUrl($b);

?>
<div class="rssIcon">
    <a href="<?=$rssUrl?>" target="_blank">
        </a>
    </div>
<link href="<?=$rssUrl?>"
    rel="alternate" type="application/rss+xml"
    title="<?=$controller->rssTitle?>" />
```

Now, each time the preceding code is executed, the custom function from `controller.php` will be called, thereby invoking the RSS functionality:

```
public function getRssUrl($b){
    $uh = Loader::helper('concrete/urls');
    if(!$b) return '';
    $btID = $b->getBlockTypeID();
    $bt = BlockType::getByID($btID);
    $c = $b->getBlockCollectionObject();
    $a = $b->getBlockAreaObject();
    $rssUrl = $uh->getBlockTypeToolsURL($bt)."/rss?bID=" .
        $b->getBlockID()."&cID=".$c->getCollectionID().
        "&aHandle=" . $a->getAreaHandle();
    return $rssUrl;
}
```

This was just one passing example of the role of tools in concrete5 block building. As you can see, the tools in concrete5 blocks are mere scripts that serve a given purpose, but owing to their usefulness, they seem to have become an integral component within the block's directory.

The MVC framework in concrete5

The model-view-controller, or MVC framework, is one of the most important aspects of concrete5. Basically, the MVC approach to block building means that most of the methods that the block employs are run from within the `controller.php` file.

Ideally, your block's controller class needs to define the following variables:

- `$btInterfaceHeight` and `$btInterfaceWidth`: These variables are used to define the width and height of your block in pixels.
- `$btTable`: This is the core block type table. This is where you will save the data for a good number of users, and this is the only variable that matters.
- `$btIncludeAll`: This is an optional variable that is useful if you want your block to be independent of concrete5's version control system.

Apart from variables, the controller class also supports a good deal of functions and methods, and we will be enumerating some of the most important ones, as follows:

Method name	Use
<code>\$controller->add()</code>	This function is automatically run when the block is added to the page.
<code>\$controller->edit()</code>	This function is automatically run when the block is being edited.
<code>\$controller->view()</code>	This function is automatically run when the block is being viewed on the page.
<code>\$controller->on_start()</code>	This function is automatically run when the user interacts with the block. Unlike the <code>_construct()</code> method, the <code>on_start()</code> method is not a resource hog.
<code>\$controller->on_page_view()</code>	This function is automatically run when the block is viewed from within a page. It comes handy when you wish to inject items to the concerned page's header from within the block.
<code>\$controller->delete()</code>	As the name suggests, this function is automatically run when a block is deleted. The purpose of this function is simply to remove data from the block's specific database table.
<code>\$controller->duplicate(\$newBlockID)</code>	This function is automatically run when the block is duplicated. A block controller can also override this function in order to provide a specific duplicate-only functionality.

Method name	Use
<code>\$controller->render(\$view)</code>	This function can be used to render a view that is in the block's directory. When passing filenames as parameters to this function, do not append the .php extension, as shown: <pre>public function view() { \$this->render("view_name"); }</pre>
<code>\$controller->save(\$args)</code>	This method inspects the columns of the block's core database table. It accepts <code>\$args</code> , which is an associated array that contains the contents of the <code>\$_POST</code> array. The <code>save(\$args)</code> method looks through the <code>\$_POST</code> array to find matching columns and saves data automatically. As such, this method does not normally need to be overridden.
<code>\$controller->set(\$key, \$value)</code>	This function is generally used within the <code>add()</code> , <code>edit()</code> , or <code>view()</code> functions. Its purpose is to take a <code>\$key</code> string and a mixed <code>\$value</code> string, and then create a variable for that name available from within the <code>add</code> , <code>edit</code> , or <code>view</code> templates.
<code>\$controller->getBlockTypeName()</code>	This function returns the name of the block type.
<code>\$controller->getBlockTypeDescription()</code>	This function returns the description of the block type.
<code>\$controller->getInterfaceWidth()</code>	This function returns the width of the block interface during the addition or deletion.
<code>\$controller->getInterfaceHeight()</code>	This function returns the height of the block interface during the addition or deletion.
<code>\$controller->addHeaderItem(\$file)</code>	This function accepts a path to a CSS or JavaScript file, and then automatically includes it when the page is executed. It can work with blocks in <code>add</code> and <code>edit</code> modes too.
<code>\$controller->getPermissionsObject()</code>	This function returns the permissions object for the given block.

Now that we have understood the major functions and properties associated with the controller class, we can get started with actual block building.

So far, we have familiarized ourselves with the directory structure of the average concrete5 block as well as the MVC framework and its associated variables and functions. We now need to get started with our first concrete5 block.

Creating a new block

Creating a new block in concrete5 can be a daunting task for beginners, but once you get the hang of it, the process is pretty simple. For the sake of clarity, we will focus on the creation of a new block from scratch. If you already have some experience with block building in concrete5, you can skip the initial steps of this section. The steps to create a new block are as follows:

1. First, create a new folder within your project's `blocks` folder. Ideally, the name of the folder should bear relevance to the actual purpose of the block. Thus, a slideshow block can be `slide`. Assuming that we are building a contact form block, let's name our block's folder `contact`.
2. Next, you need to add a controller class to your block. Again, if you have some level of expertise with concrete5 development, you will already be aware of the meaning and purpose of the controller class. We discussed the role and use of the controller class earlier in this chapter, so there is not much to worry about here. Basically, a controller is used to control the flow of an application, say, it can accept requests from the user, process them, and then prepare the data to present it in the result, and so on.
3. For now, we need to create a file named `controller.php` in our block's folder. For the contact form block, this is how it is going to look (don't forget the PHP tags):

```
class ContactBlockController extends BlockController {
    protected $btTable = 'btContact';
    /**
     * Used for internationalization (i18n).
     */
    public function getBlockTypeDescription() {
        return t('Display a contact form.');
```

```
    }
```

```
    public function getBlockTypeName() {
        return t('Contact');
```

```
    }
```

```
    public function view() {
        // If the block is rendered
    }
```

```
    public function add() {
        // If the block is added to a page
    }
```



```
public function edit() {  
    // If the block instance is edited  
}  
}
```



The preceding code is pretty simple and seems to have become the industry norm when it comes to block creation in concrete5.

Basically, our class extends `BlockController`, which is responsible for installing the block, saving the data, and rendering templates. The name of the class should be the Camel Case version of the block handle, followed by `BlockController`.

We also need to specify the name of the database table in which the block's data will be saved. More importantly, as you must have noticed, we have three separate functions: `view()`, `add()`, and `edit()`. The roles of these functions have been described earlier.

4. Next, create three files within the block's folder: `view.php`, `add.php`, and `edit.php` (yes, the same names as the functions in our code). The names are self-explanatory: `add.php` will be used when a new block is added to a given page, `edit.php` will be used when an existing block is edited, and `view.php` jumps into action when users view blocks live on the page.

Often, it becomes necessary to have more than one template file within a block. If so, you need to dynamically render templates in order to decide which one to use in a given situation. As discussed in the previous table, the `BlockController` class has a `render($view)` method that accepts a single parameter in the form of the template's filename. To do this from `controller.php`, we can use the code as follows:

```
public function view() {  
    if ($this->isPost()) {  
        $this->render('block_pb_view');  
    }  
}
```

In the preceding example, the file named `block_pb_view.php` will be rendered instead of `view.php`.

To reiterate, we should note that the `render($view)` method does not require the `.php` extension with its parameters.

Now, it is time to display the contact form. The file in question is `view.php`, where we can put virtually any HTML or PHP code that suits our needs. For example, in order to display our contact form, we can hardcode the HTML markup or make use of Form Helper to display the HTML markup. Thus, a hardcoded version of our contact form might look as follows:

```
<?php defined('C5_EXECUTE') or die("Access Denied.");
global $c; ?>
<form method="post" action="<?php echo $this->action('contact_submit');
?>">

    <label for="txtContactTitle">SampleLabel</label>
    <input type="text" name="txtContactTitle" />
    <br /><br />
    <label for="taContactMessage"></label>
    <textarea name="taContactMessage"></textarea>
    <br /><br />
    <input type="submit" name="btnContactSubmit" />
</form>
```

Each time the block is displayed, the `view()` function from `controller.php` will be called. The `action()` method in the previous code generates URLs and verifies the submitted values each time a user inputs content in our contact form.

Much like any other contact form, we now need to handle contact requests. The procedure is pretty simple and almost the same as what we will use in any other development environment. We need to verify that the request in question is a `POST` request and accordingly, call the `$post` variable. If not, we need to discard the entire request. We can also use the mail helper to send an e-mail to the website owner or administrator.

Before our block can be fully functional, we need to add a database table because `concrete5`, much like most other CMSs in its league, tends to work with a database system.

In order to add a database table, create a file named `db.xml` within the concerned block's folder. Thereafter, concrete5 will automatically parse this file and create a relevant table in the database for your block. For our previous contact form block, and for other basic block building purposes, this is how the `db.xml` file should look:

```
<?xml version="1.0"?>
<schema version="0.3">
  <table name="btContact">
    <field name="bID" type="I">
      <key />
      <unsigned />
    </field>
  </table>
</schema>
```

You can make relevant changes in the preceding schema definitions to suit your needs. For instance, this is how the default YouTube block's `db.xml` file will look:

```
<?xml version="1.0"?>
<schema version="0.3">
  <table name="btYouTube">
    <field name="bID" type="I">
      <key />
      <unsigned />
    </field>
    <field name="title" type="C" size="255"></field>
    <field name="videoURL" type="C" size="255"></field>
  </table>
</schema>
```

The preceding steps enumerate the process of creating your first block in concrete5. However, while you are now aware of the steps involved in the creation of blocks and can easily work with concrete5 blocks for the most part, there are certain additional details that you should be aware of if you are to utilize the block's functionality in concrete5 to its fullest abilities. The first and probably the most useful of such detail is validation of user inputs within blocks and forms.

Validating user inputs

Now that we have learned how to create blocks and have also gained a good understanding of the MVC approach in concrete5, the next step is to validate user inputs because in all likelihood, the most important blocks are ones that seek user inputs, generally in forms such as the contact form that we created previously.

As is obvious, when it comes to validating user inputs within the browser, we can rely on HTML5 and jQuery. In the next example, I will be using e-mail inputs for demonstration purposes, but you can, of course, remodel to input other fields, such as URLs, numbers, and so on. Note that validation on the client side is generally considered not secure and you might be better off validating the user input coming from the server.

The user input in HTML5 is not rocket science, thankfully. There is a vast range of input types that we can make use of, and the best part about HTML5 is that even if a particular web browser does not support a given input type, it can still fall back on treating that input type as text. Thus, as a developer, when working with HTML5 input types, you need not be worried about fallback mechanisms for web browsers that might not support all the given input types.

Speaking of concrete5, form elements that validate user inputs are generally coded directly using HTML or via the concrete5 Form Helper. Since this is the standard practice irrespective of the CMS you are working on, there are chances that you might have seen and used the given code to validate the user inputs. The basic code looks something like the following:

```
<input type="text" name="user_email_address">
<?php
$form = Loader::helper('form');
echo $form->text('user_email_address');
?>
```

Before we move any further, we need to create HTML5 form elements from PHP. For HTML5 browsers, the easiest validation solution is to change the form elements to e-mail input types by recoding the relevant PHP lines to give out an e-mail input. There are two methods to do this, and we will discuss both of them. Also, note that the parameters and actual methods are almost the same for the text input elements as well, so you can reuse this example with the same parameters.

We can modify form elements for HTML5 with jQuery. In general, there are two justifications for using this method: you are more comfortable using jQuery rather than modifying the actual input form via concrete5, and/or you are using an older version of concrete5 (5.4.1.1 or earlier) and are thus unable to directly modify the actual form.

In any case, for our purpose, we can use the following code to modify all the text elements that carry `email` in their name to e-mail elements that are compatible with HTML5:

```
<script type="text/javascript">
    $(document).ready(function() {
        $('input[type="text"][name*="email"]').
            add('input[type="text"][name*="Email"]').each(function()
            {
                var input_html = $('<div>').append( $(this).
                    clone() ).html();
                var better_html =
                    input_html.replace(/type="text"/i, 'type="email"');
                $(this).replaceWith(better_html);
            });
    });
</script>
```

What does the previous code do? Basically, the jQuery selector searches for text inputs that carry `email` somewhere in their name. Now, since we cannot outright change the input type attribute, we have decided to clone the existing input and place it inside a `div` element in order to extract the HTML of the element as text. Then, we simply replace `type="text"` with `type="email"`, and then replace the original input element with the modified HTML.

Using the previous code, web browsers that support the HTML5 e-mail element can validate the e-mail element before the form is submitted.

Note that since jQuery selectors are case-sensitive, we need to look for both `email` and `Email`. Also, in place of the previous selector, you can use other jQuery expressions and filters as well.

Thus, in this method, since we cannot change the type of an input element, we have decided to receive the HTML as text, modify it, and then write it back.

Another method is to validate the data using jQuery without actually using HTML5. While HTML5 has almost become the standard norm and every modern web browser supports it well, you can still choose to reduce your dependency on HTML5 and rely on jQuery plugins that can offer comprehensive form validation (one such plugin is `jquery.validate.js`, which we will be using next).

In order to use any jQuery plugin, you first need to copy it to your concrete5 installation server, and then add it to your page using `addHeaderItem()`. Our `jquery.validate.js` plugin attaches itself to a form, and then works by searching for classes that are associated with input elements. Thus, instead of replacing the text elements that carry a given string in their name, say, `email` from the previous example, our plugin can add the `required` and `email` classes to the elements in a straightforward manner, as follows:

```
<script type="text/javascript">
    $(document).ready(function()
    {
        $('input[type="text"][name*="email"]').
            add('input[type="text"][name*="Email"]').each(function(){
                $(this).addClass('required email');
            });
        $('form').validate();
    });
</script>
```

In the preceding code, the `$('form')` selector is used to attach the `validate()` method to the form. For complex setups with multiple forms on the same page, you will need to modify this selector accordingly.

The `contact_submit` method, as described in `view.php`, will be used to submit the contact form.

In the previous examples, we discussed two simple ways to validate user inputs in concrete5—first, by recoding the form in PHP, and secondly, by modifying the form using jQuery. You can use similar techniques to validate input elements other than e-mails, such as URLs, phone numbers, names, and so on. Keep in mind, though, that the client-side validation is not secure; it is wiser to use the `Validate` method of the controller.

Summary

In this chapter, we familiarized ourselves with concrete5 blocks. We covered the directory structure of blocks as well as the importance and usage of the MVC framework and the controller class. Of course, we learned how to create our very first block in concrete5 and the ways in which we can validate user input in order to make our blocks and forms useful for the end user.

In the next chapter, we will be taking our adventures with concrete5 blocks to another level. We will discuss advanced concrete5 blocks and add-ons. Since your main focus as a developer will be to either build websites for clients using concrete5 or to offer specialized concrete5 services, we will be talking at length about packaging advanced blocks into concrete5 add-ons that are ready for distribution. Beyond this, we will also talk about database tables, file management, and other relevant information in order to help you get the most out of concrete5 blocks and add-ons.

3

Advanced concrete5 Blocks

In the previous chapter, you gained an understanding of how concrete5 blocks work. We discussed the directory structure of concrete5 blocks and the MVC framework. You also learned how to create blocks in concrete5.

Moving on, in this chapter, we will further focus on building blocks from an advanced perspective. Here are some of the major topics that will be covered in this chapter:

- Using database tables
- Advanced concrete5 blocks
- File management in concrete5 blocks
- Overriding the default functionality of concrete5
- Packaging your code for distribution

So, without wasting any more time, let's get started with advanced block development in concrete5.

Using database tables

As is common knowledge, concrete5 stores its data in databases. So, if you are keen on working with and mastering concrete5, you will need to learn how to get and manipulate the data from database tables. In fact, the database forms an integral part of concrete5 (or any similar database-driven CMS, for that matter).

However, if your intent is just to access data or perform some basic operations (such as the creation of a basic block, which we mastered in the last chapter of this book), concrete5 provides enough methods to safely get the job done without actually having to bother with database queries and connections. Yet, if you need to perform complex and advanced functions, such as the creation of an advanced block, mastering the concrete5 database is an important task that you should accomplish.

You should note that when building a block, your calls to the database should generally be contained within the controller files. As such, it is advisable to keep the presentation layer separate from database connections.

In fact, in proper terms, you can place your commands in order to access the database almost anywhere. However, if you follow the MVC protocol, you will need to collect all functions that directly access the database. Going by this protocol, it is a good practice to place such functions in a page or block's `controller.php` file. You should consider moving the bulk of your PHP code to the controller files. As a rule, the less PHP code you have in View, the better it is!

First, let's begin with establishing connections to and querying the database.

Database connections

As soon as you install concrete5, a file is automatically created, and it contains the database connection information for the concerned concrete5 installation. It contains information about the server, the name of the database, the database username and password, and so on as constants. It acts as the de facto settings file that concrete5 uses to connect to the database. It is located at `your-site-path/config/site.php`. The `config/site.php` file might also contain other site configuration, such as white-labeling-related constants and various things such as `ENABLE_NEWSFLOW_OVERLAY`.

The creation of such a file means that you can forget about boring tasks such as the storage of database login credentials and so on, since concrete5 does all this for you. So, you can simply connect to the database and start interacting with the object.

Now, in order to connect to the database (not through the user, though), we need to load the active database object. In order to do so from inside any function or class, the following command can be used:

```
$db= Loader::db();
```

Using this command, the database object becomes active and available in the current scope.

If, however, you ever need to connect to different or multiple databases, you will be required to switch the active database object. Here is how we do this:

```
$db = Loader::db( 'nameofserver', 'nameofuser', 'password',  
    'namedatabase', true);
```

Once you're done working with the different databases, we come back to the default database and reload our default database object. At this point, users need to create new ADODB connections.

Database queries

Now that you have learned how to establish a database connection, it is time to query the database. After loading the database object, we can fetch data using the `Execute` method and passing the database query as an argument. Here is how this works:

```
$db->Execute('select * from TableName');
```

As is obvious, the preceding example selects everything from the concerned table.

In order to simplify database queries, `concrete5` uses the `ADODB` database extraction layer. Basically, the `ADODB` database extraction layer provides us with a simple way to use queries with a shortened syntax. Consider this to be a type of database middleman that intends to simplify your interaction with the database. In fact, `ADODB` has several benefits of its own:

1. Firstly, it allows your application to be used on a number of database types and not just MySQL. If you are just getting started with database queries, it might be that you do not need and/or know much about databases other than MySQL (which is alright and will serve your purpose, since MySQL is very popular and nearly every platform supports it).
2. Secondly, `ADODB` comes with the `AutoExecute` `INSERT` and `UPDATE` methods. For a developer who is just getting started with database queries, `ADODB`'s `AutoExecute` methods mean that you are saved from the possibility of SQL injections and code hacks, which would otherwise arise if one were to manually write hundreds of queries. Thus, `ADODB` can make your application more secure, assuming users call code as per `ADODB` norms.
3. Lastly, `ADODB` makes it very easy to grab data from the database. It returns the data in an associated array that has the `($arrayname[$column_name] = $value)` form. Note that numeric fields are returned as strings.

Of course, you can still use the longer, traditional version of queries (more like the execution method example shown previously), but it is advisable and wiser to employ `ADODB` whenever possible, simply because this technique is way more flexible, straightforward, secure, and keeps your data tidy.

Let's say we want to return all rows from a given table. Using the `ADODB` database extraction layer model, our input would be as follows:

```
$db->GetAll("select col from table");
```

The preceding method will return all rows from the concerned table as two-dimensional arrays.

Getting data from the database

As a developer working with concrete5, there is a good chance that you will be required to pull data from the database rather than actually writing it to the database manually. So, we will focus on some of the popular ways of asking the database for the data.

Let's suppose that we have this sample database table named `trainTBL` to work on:

sbID	Name	Orgn	Destination
1	Pelham123	Pelham	NewYork
2	SiberiaXpress	Siberia	Moscow
3	AnkaraMail	Izmir	Ankara
4	NewIzmirRail	Izmir	Istanbul

The GetOne command

The `GetOne` command is used when you wish to return just a single value. For example, if you wanted to write a query that would inform you about the destination of a particular train, here is how to do this:

```
$trainName = "Pelham123";  
$destination = $db->getOne("SELECT destination FROM trainTBL WHERE  
    name LIKE '?' " , array($trainName) );
```

After running the preceding command, the value of `$destination` will be set to the string `NewYork`, thereby telling us that the destination of `Pelham123` is `NewYork`.

This function takes two arguments. The first one is the general query, with question marks where the variable data goes. The second one is an input array that is used to replace the question marks successively.

We can also simplify the previous command as follows:

```
$trainName = "Pelham123";  
$destination = $db->getOne("SELECT destination FROM trainTBL WHERE  
    name LIKE '$trainName' ");
```

However, this would make SQL injection attacks vulnerable if the `$trainName` value was sent from the browser as the `$POST` or `$GET` variable. So, it is safer to stick with the original command definition.

The GetRow command

The second command in our list, which is `GetRow`, is used when you wish to retrieve all of the fields in a matching row. Much like the `GetOne` command, `GetRow` is useful when you want just a single match to your query.

Here is an example:

```
$sbID = 2;
$trainData = $db->GetRow("SELECT * FROM trainTBL WHERE sbID =
    ?" , array($sbID));
```

Thereafter, the resultant `$trainData` variable will be an associative array, as follows:

```
$trainData['sbID'] = 2
$trainData['name'] = "SiberiaXpress"
$trainData['origin'] = "Siberia"
$trainData['destination'] = "Moscow"
```

Now, in order to display the data from this associative array, we can employ a simple loop:

```
foreach ($trainData as $column => $value) {
    echo "$column = $value <br />";
}
```

The preceding code will print the following:

```
sbID = 2
name = SiberiaXpress
origin = Siberia
destination = Moscow
```

Now, on to the third command.

The GetAll command

As the name suggests, the `GetAll` command is deployed when you wish to return one or more rows of data.

Let's look at an example:

```
$origin = "Izmir";
$strTrains = $db->GetAll("SELECT * FROM trainTBL WHERE origin LIKE
    '?' ", array($origin));
```

The preceding command creates a multidimensional array that is numerically indexed for each matching row (and has a nested associative array within itself). Our `$trTrains` variable will look as follows:

```
$trTrains[0]['sbID'] = 3
|  |
| --- |
| $trTrains[0]['name'] = "AnkaraMail" |
| $trTrains[0]['origin'] = "Izmir" |
| $trTrains[0]['destination'] = "Ankara" |
| $trTrains[1]['sbID'] = 4 |
| $trTrains[1]['name'] = "NewIzmirRail" |
| $trTrains[1]['origin'] = "Izmir" |
| $trTrains[1]['destination'] = "Istanbul" |

```

Again, to print our data, we will use a simple loop:

```
foreach ($trTains as $trainData) {
    foreach ($trainData as $column => $value) {
        echo "$column = $value <br />";
    }
    echo "<hr />";
}
```

This is what our output will look like:

```
sbID = 3
name = AnkaraMail
origin = Izmir
destination = Ankara

scid = 4
name = NewIzmirRail
origin = Izmir
destination = Istanbul
```

This covers the basic and most commonly used methods of fetching data from database tables. If you wish to learn more about such methods and commands, you can always refer to the ADODB PHP library at <http://phplens.com/lens/adodb/docs-adodb.htm>.

The database debugging mode

You can use the database debugging mode when troubleshooting your queries. The debug mode simply lets you see all your queries printed inline in the page. To activate the debug mode, just add this line to an element from within the View layer:

```
Database::setDebug(true);
```

To deactivate the debug mode, add the following line:

```
Database::setDebug(false);
```

Working with advanced concrete5 blocks

We already discussed the creation of basic concrete5 blocks at length in the last chapter. Furthermore, we also talked about concrete5 tools and other relevant details that you need to know in order to get started with block development. Therefore, in this chapter, we will now focus on certain advanced concepts in detail, which will come in handy for more detailed and complex block development in concrete5.

If you are focusing on user interaction and an overall improved experience, you will need to create advanced or more complex blocks instead of simple and straightforward basic blocks. The techniques and methods of block creation remain the same, but there are certain additional tips and usage scenarios that you need to be aware of in order to efficiently create concrete5 blocks and use them to your advantage.

The composer

The composer allows certain pages to be created from within the Dashboard rather than the in-page editor. When setting up defaults for a composable page, your blocks will need a special view in order to be correctly added and edited from within the composer interface.

The basic difference here is that when you add or edit a block using the in-page editor interface, you submit changes to the concerned block only. However, in the case of the composer interface, you submit changes to multiple blocks. Quite obviously, you will need to name the form fields in this case in a manner that they remain distinguishable from one another.

The direct method of making a block composer friendly is as follows:

```
//Composer-ready:
<textarea name="<?php echo $this->field('content') ?>">
//Normal:
<textarea name="content">
```

The `field` function in the preceding example returns the field example in a way that it remains identifiable in the `POST` request.

Working with file management

Let's now focus on how to add the file manager to our blocks. Including the file manager in blocks is pretty easy, and by adding the following code, you will be able to create an interface that uses the concrete5 file manager:

```
$bt = Loader::helper('concrete/asset_library');
echo $bt->file('optional-ID', 'fID', t('Get a file.'), $ff,
    $args);
```

In the preceding example, we are using an asset library helper in order to bring up the file manager. Here is an overview of the arguments specified:

- `$id`: This string is used to create an ID for the HTML elements
- `$postname`: This string will be used to reference the file ID and save it in the block's controller
- `$text`: This string will display the text when the user is asked to pick the file
- `$ff`: This is a file object; generally, this is the file the block is already using
- `$args`: This is a set of attributes; it is a keyed array of the attributes' handle (key) and value

The last option might confuse you a little, since it deals with attributes. We will, however, turn to concrete5 attributes and their mode of functioning in the next chapter of this book, so you need not worry much about it right now.

Overriding the default functionality of concrete5

Sometimes, it is not possible to simply tweak a given content type that is included in a package or in the core itself as and when we want. In such cases, we can use the override feature of concrete5 to suit our needs.

By using overrides in concrete5, we basically change a part of the core CMS or an add-on without actually modifying the files in the core or add-on folder. This enables us to update our installation whenever required, without losing any of the changes or overrides that we had made.

Once you create an override in concrete5 and log in to the concrete5 Dashboard thereafter, you can look at the list of all the overrides that you have created in the website environment section on the Dashboard. This enables you to have an overview of all the overrides that you have made and gives you information on the changes at a glance.

How to override?

Overriding the default functionality of concrete5 is pretty simple. Basically, you just need to copy the concerned file right up to its equivalent path in the concrete5 root directory structure. After that, concrete5 will use the new file instead of the one in its default location. concrete5 also has an override cache, so you might not see all your changes right away. In this case, it is advisable to turn off caching when entering the development mode.

This model works for nearly all overrides, excluding a few exceptions, which are mentioned as follows:

- Themes cannot be overridden by copying them from the packaged directory to the `/themes/` folder. However, you can install the theme as a new theme by modifying its `description.txt` file (refer to *Chapter 1, Building a Concrete 5 Theme*).
- If you are overriding a packaged tool, you will need to move that tool to the folder with the add-on's name. Thereafter, once you have moved the tool to a folder with the given add-on's name, you can override the packaged tool.
- In addition, if you are copying `model/library/...`, the contents of the class will be blank. Thus, you need to add the contents of the class after copying `model/library` in order to override the default functionality. You can then add new functions as per your needs (this does not apply to versions of concrete5 that are older than 5.6, though).

Thus, as an example, if we were to copy the `public_html/concrete/single_pages/login.php` file and place it at `public_html/single_pages/login.php`, the override mechanism would use the latter file instead of the former.

Packaging your code for distribution

It is a standard practice to bundle your code together for distribution, installation, and uninstallation. For this purpose, the concrete5 package format is the way to go. While the package format is generally used by the concrete5 marketplace, it can also be used for independent distribution and installation.

You can bundle a lot many things using the concrete5 package format. For example, you can include libraries, elements, tools, block types, page types, attribute type, attribute keys, single pages, dashboard modules, themes, translations/localizations, and several other components within packages. We already discussed how to package a concrete5 theme and make it ready for distribution in *Chapter 1, Building a Concrete 5 Theme*. Here, we will take a closer look at the concrete5 package structure and installation details.

Packages in concrete5

By means of packages, you can make components easily reusable and installable across multiple concrete5 websites. Be it themes, templates, or blocks, you can package together any type of components as per your needs.

Let's take a look at the directory structure of a typical concrete5 package. In terms of files, in the root folder of the package, `controller.php` is the most important file. As mentioned in other chapters, there are other files as well, such as `icon.png` (a 97 x 97 pixels icon image) and `changelog.txt`. However, speaking of folders, the folder structure of certain packages might include other relevant folders too as per the needs. For example, an application's package will need folders for CSS and JS elements as well as page types, and so on.

The following screenshot shows you the packages in concrete5:

samp_package	1 item folder
blocks	4 items folder
samp_block	6 items folder
add.php	0 bytes PHP script
controller.php	0 bytes PHP script
db.xml	0 bytes XML document
edit.php	0 bytes PHP script
icon.png	0 bytes PNG image
view.php	0 bytes PHP script
changelog.txt	0 bytes plain text document
controller.php	0 bytes PHP script
icon.png	0 bytes PNG image

The package controller

As already stated, each package has a `controller.php` file in its base directory. This file contains the information that is required in order to identify the concerned package and install it correctly.

Here is how a `controller.php` file generally looks:

```
defined('C5_EXECUTE') or die(_("Access Denied."));
class RandomPkgPackage extends Package {
    protected $pkgHandle = 'random_pkg';
    protected $appVersionRequired = '5.3.0';
    protected $pkgVersion = '1.0';
    public function getPackageDescription() {
        return t("An example of package in c5");
    }
    public function getPackageName() {
        return t("Random Pkg");
    }
    public function install() {
        $pkg = parent::install();
        // install block
        BlockType::installBlockTypeFromPackage('random_pkg', $pkg);
    }
}
```

It is important to make note of certain key points here:

- The name of the controller class contains the uppercase version of the package handle with `Package` appended at the end. In our example, `random_pkg` becomes `RandomPkgPackage`. Notice that there are no spaces.
- If you wish to add your package to the concrete5 marketplace, make sure that it has a unique handle. Also, if it is a theme's package, you should start it with `theme_`.
- The `Package::install()` method is required in nearly all packages excluding templates. Its definition depends on what our package intends to install (for instance, in the previous example, we are installing a block).
- The `upgrade()` method too is used in nearly all packages and is, in fact, as important as the `install()` method.
- You can also use the `InstallController::configure()` method for advanced and complex package installations.

Summary

This brings us to the end of this chapter. So far, we have covered details such as database operations, especially getting data from the database, database debugging, and database connections.

We also focused on advanced concrete5 blocks. We learned how to work with file management in concrete5 blocks and also mastered the composer view for our blocks. Beyond this, overriding the default functionality of concrete5 was also discussed.

Lastly, we talked about concrete5 packages, with special focus on the package controller.

Having completed the block development, we will be focusing on attributes and attribute types in the next chapter. We will also learn how to work with custom attribute types in concrete5.

4

Attributes in concrete5

In the previous chapter, we learned about advanced concrete5 block development. We covered topics such as file management in concrete5, the use of database tables in block development, and so on.

Furthermore, we also learned how to override the default functionality of concrete5, and package our code for distribution.

Thus, by now, we have mastered both basic and advanced block development, and also learned how to work with and extract data from concrete5 databases.

Moving on, in this chapter, we will be taking a look at concrete5 attributes and attribute types. Here is a quick rundown of what this chapter talks about:

- What are attributes in concrete5?
- An attribute category, type, and key
- Creating and dealing with custom attribute types

Before moving any further, we need to first answer a basic question: what are attributes in concrete5?

An overview of the concrete5 attributes

In simple terms, attributes are used in concrete5 to bind data to given items – be it pages, users, or files. Much like blocks, attributes have types and can store different data depending on the basis of the concerned type.

What makes an attribute different from a block? Well, firstly, unlike blocks, attributes are actually available for files, users, and pages. Next, attributes can also be used for custom objects.

Another obvious benefit of attributes is the fact that they are built within the concrete5's searching and listing frameworks—you can easily search and sort objects on the basis of the attached attribute value.

Thus, in concrete5 terminology, an attribute is some data that you wish to store. It can be something as simple as a short text field, a radio button, a checkbox, an image, or a file, and so on. Of course, attributes can also contain much more complex data.

Attributes may or may not be displayed to the end users, and instead used by developers to collect and store data. Thus, if blocks deal with the frontend, attributes deal with the backend.

So, where does one use attributes?

The simple answer is wherever and whenever you wish to collect, store, search, and manipulate data.

Attributes can be assigned to pages, files, users, and other types of data. Furthermore, you can even create your own custom attribute types (this is something that we will explore later in this chapter).

Now, before we get started with working with attributes, we need to be clear about certain terms and concepts related to concrete5 attributes.

Attribute terminology in concrete5

Before going any further, we will take a quick look at the common terminology associated with attributes in concrete5. We will be referring to and using these terms every now and then in this chapter, so it is a good idea to grasp them properly before going any further:

- **Attribute categories:** An attribute category is the type of object that the given attribute is applied to. In general, the default categories implemented by concrete5 for attributes are `Page`, `File`, and `User`. These categories can suffice for most developers, especially if you are getting started with concrete5 development. However, new categories can be created in case of additional add-ons, but note that the creation of new categories of databases involves changes that may or may not impact the performance of your concrete5 installation in the long run. Thus, it can be said that the category of an attribute is the object type that the given attribute is bound to. Let's look at the attributes such as `Width` and `Height` for a given file. These will obviously be a part of the `File` attribute category.

- **Attribute types:** An attribute type refers to the type of object that the attribute is applied to. Whenever you create an attribute for a specific purpose, you can choose an attribute type for it. For example, `text_area` is an attribute type signifying that the given page might have several attributes of this type. Basically, the type of an attribute determines the manner in which the given attributes stores, presents, and works with data. If you intend to extend the default concrete5 attribute system and/or possibly store new types of data, you will need to attain very good knowledge of attribute types, and thereby, create custom attribute types as and when necessary.
- **Attribute keys:** An attribute key, in concrete5 terminology, is an object that holds information about a particular attribute. Each time you create a new attribute, you are also automatically creating an `AttributeKey` object.

Now that we have understood the basic meaning of each major term associated with concrete5 attributes, it is time to dig deeper and gain better insight into the function and usage of each of these terms: keys, types, and categories.

Understanding and working with attribute terminology in concrete5

We will now look at attribute categories, attribute keys, and attribute types in an detailed manner to help you understand their role in relation to concrete5 attributes. Soon after, we will move on to the creation of custom attribute types.

The attribute category

concrete5 has three major attribute categories:

- `File`
- `Page`
- `User`

If you are adding attribute functionality to your own object, you will need to work with attribute categories. Otherwise, for all practical purposes, as a developer you might not really need to directly interact with the `AttributeCategory` class. However, for the sake of clarity, I have discussed the creation of your own attribute category in the following section.

Creating an attribute category

As already mentioned, if you are adding attribute functionality to your objects, you will need to create an attribute category for such objects. The process is simple as you basically need to add data to the database.

Let's say you have a custom object named `something` and you need to add an attribute category for this object. Run the following command:

```
$allowSets = false;
AttributeKeyCategory::add('something', $allowSets);
```

That's all. If, however, your attribute category needs to be added to a package as well, you will need to run `$pkg` as well, as follows:

```
$allowSets = false;
AttributeKeyCategory::add('something', $allowSets, $pkg);
```

The preceding code will create an attribute key category for your object. Now, you can create attribute key objects against this category.

The next step in this process is to create the class that extends the `AttributeKey` and `AttributeValue` objects. We will discuss this step in the next section, when we talk about attribute keys.

The attribute key

As already discussed earlier in this chapter, attribute keys contain bits of data that can be stored against data types. However, before you can add attribute keys from the concrete5 dashboard and start storing data, you need to create attribute categories pertaining to relevant attribute keys.

Let's try to understand attribute keys in concrete5 with the help of an example. Let's say we have a custom object at hand (say, a widget or any such object), and we wish to use concrete5 attributes against the given object. For this purpose, we have already created an attribute key category for our object (on the basis of the example shown in the *The attribute category* section). Now, in order to use concrete5 attributes against the given object, this is how we should proceed:

1. First, we need to create database tables to store the relevant data.
2. Secondly, we need to create a class to extend core classes such as `AttributeKey` and `AttributeValue`.
3. Lastly, in order to add our attribute keys, we need to create a database interface.

Creating a database file for the attribute keys

Now, at this point, we need to create a database table for the concerned attribute values of the object in question. The purpose of our table is to accurately bind the attribute value to the object. The variable that holds the information for the attribute value (`avID` in the following example) should be an unsigned integer. We will use the temporary term `objct` to signify the position of the object's name, though of course you need to replace that with the relevant term when implementing the code.

Here is an example:

```
create table ObjctAttributeValues
(
    objctID int unsigned not null default 0,
    akID int unsigned not null default 0,
    avID int unsigned not null default 0,
    primary key (objctID, avID, akID)
);
```

The preceding table is pretty simple. The `objctID` parameter serves as the primary key of the concerned object; whereas `akID` is the primary key of the `ObjctAttributeKey` object, and `avID` is the primary key for the `ObjctAttributeValue` object. Tables should be created via an XML file when you install the package.

Creating a class file

Now, it is time to create the class file. Since our class will be derived from `AttributeKey`, we will place it at the directory path `models/attribute/categories/{ATTRIBUTE_CATEGORY_HANDLE}.php`, which is relative to the root folder of our `concrete5` installation or the package directory.

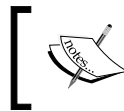
Our file needs to contain two classes: `ObjctAttributeKey` and `ObjctAttributeValue`, both of which will extend `AttributeKey` and `AttributeValue`, respectively. Let's look at the contents of each of these classes.

The ObjectAttributeKey class

The ObjectAttributeKey class consists of the following methods and properties:

- `$searchIndexFieldDefinition`: This defines the data type for the primary key in the search index table. Since it deals with the primary key and table, it uses the ADODB data dictionary format (described in the previous chapter). For example, the following line defines the primary key of the indexed search table as `abcdID`, an unsigned integer:

```
protected $searchIndexFieldDefinition = 'abcdID I(11) UNSIGNED
NOTNULL DEFAULT 0 PRIMARY';
```



It is worth noting that if your attribute key category is not searchable, you can (and should) omit this part.

- `getIndexedSearchTable()`: This method returns the name of the indexed search table that holds the attributes. Once again, if your attribute key category isn't searchable, you can (and should) omit this as well.
- `getAttributes($primaryKey, $method = 'getValue')`: This method accepts the initial arguments passed to it and then binds the attribute value to an object. As such, it needs to fetch all the attributes for the concerned object that the attribute key category is going to bind itself to. For users, this method looks something like this:

```
getAttributes($uID, $method = 'getValue')
```

And for pages, we use the following:

```
getAttributes($pageID, $pageVersionID, $method = 'getValue')
```

- `getColumnHeaderList()`: This function will return the following:

```
return parent::getList('object', array('akIsColumnHeader' => 1));
```
- `getSearchableIndexedList()`: This function will return the following:

```
return parent::getList('object', array('akIsSearchableIndexed' => 1));
```
- `getSearchableList()`: This function will return the following:

```
return parent::getList('object', array('akIsSearchable' => 1));
```

- `getAttributeValue($avID, $method = 'getValue')`: This method will implement the following:

```
$av = ObjectAttributeValue::getByID($avID);
$av->setAttributeKey($this);
return call_user_func_array(array($av, $method), array());
```
- `getByID($akID)`: This method will implement the following:

```
$ak = new ObjectAttributeKey();
$ak->load($akID);
if ($ak->getAttributeKeyID() > 0)
{
    return $ak;
}
```
- `getByHandle($akHandle)`: This method queries the database for the ID of the matching handle.
- `getList()`: This method returns the list.
- `saveAttribute($object, $value = false)`: This method implements the `getAttributeValueObject` call, and then runs the `saveAttribute` function.
- `add($type, $args, $pkg = false)`: This can be used to add special information to the attribute category in the `ObjectAttributeKeys` table.
- `update($args)`: Apart from adding special information to the attribute category in the `ObjectAttributeKeys` table, it can also be used to run `parent::update($args)`.
- `delete()`: After running `parent::delete()`, it will make use of the following command:

```
$db = Loader::db();
$db->Execute('delete from ObjectAttributeKeys
    where akID = ?', array($this->getAttributeKeyID()));
$r = $db->Execute('select avID from ObjectAttributeValues
    where akID = ?', array($this->getAttributeKeyID()));
while ($row = $r->FetchRow())
{
    $db->Execute('delete from AttributeValues
        where avID = ?', array($row['avID']));
}
$db->Execute('delete from ObjectAttributeValues
    where akID = ?', array($this->getAttributeKeyID()));
```

The ObjectAttributeValue class

The ObjectAttributeValue class consists of the following methods and properties:

- `set*`: You will need to create a setter for the object that you are binding the attribute to. For example:

```
public function setObject($object)
{
    $this->object = $object;
}
```

- `getByID($avID)`: This method provides the ID for the attribute value. Here is how a sample definition will look:

```
$pav = new ObjectAttributeValue();
$pav->load($avID);
if ($pav->getAttributeValueID() == $avID)
{
    return $pav;
}
```

- `delete()`: While its usage is obvious, before we run `delete()` on the parent object, you need to ensure that the attribute's value is not being referenced anywhere else in the concerned table. Thereafter, this is how we implement it:

```
$db = Loader::db();
$db->Execute('delete from ObjectAttributeValues
    where yourKeyID = ? and akID = ? and avID = ?', array(
    $this->object->getObjectID(),
    $this->attributeKey->getAttributeKeyID(),
    $this->getAttributeValueID()
));

$num = $db->GetOne('select count(avID) from
    ObjectAttributeValues where avID = ?', array($this-
    >getAttributeValueID()));
if ($num < 1)
{
    parent::delete();
}
```

Creating a view page for the attribute category

At this point, you will require an interface to create new attributes. You can do so from the concrete5 dashboard itself; just make sure that you include the `attribute/type_form_required.php` element and the `dashboard/attributes_table.php` element.

On the other hand, if you intend to extend the existing attribute types into the custom attribute types, read on, for that is discussed in the *Using the new custom attribute type* section of this chapter.

Adding attribute capabilities to the objects

Attributes can and should be added to objects that they support. The following example can be used to add to the supporting objects (replace `objct` in the given code with the concerned object):

```
public function setAttribute($ak, $value)
{
    Loader::model('attribute/categories/objct');
    if (!is_object($ak))
    {
        $ak = ObjectAttributeKey::getByHandle($ak);
    }
    $ak->setAttribute($this, $value);
    $this->reindex();
}
public function reindex()
{
    $attrs = ObjectAttributeKey::getAttributes
        ($this->getObjectID(), 'getSearchIndexValue');
    $db = Loader::db();
    $db->Execute('delete from ObjectSearchIndexAttributes
        where objctID = ?', array($this->getObjectID()));
    $searchableAttributes = array('objctID' =>
        $this->getProductID());
    $rs = $db->Execute('select * from ObjectSearchIndexAttributes
        where objctID = -1');
    AttributeKey::reindex('ObjectSearchIndexAttributes',
        $searchableAttributes, $attrs, $rs);
}
public function getAttribute($ak, $displayMode = false)
{
    Loader::model('attribute/categories/objct');
    if (!is_object($ak))
    {
        $ak = ObjectAttributeKey::getByHandle($ak);
    }
}
```

```
    }
    if (is_object($ak))
    {
        $av = $this->getAttributeValueObject($ak);
        if (is_object($av))
        {
            return $av->getValue($displayMode);
        }
    }
}

public function getAttributeValueObject($ak,
    $createIfNotFound = false)
{
    $db = Loader::db();
    $av = false;
    $v = array($this->getProductID(), $ak->getAttributeKeyID());
    $avID = $db->GetOne("select avID from ObjectAttributeValues
        where objectID = ? and akID = ?", $v);
    if ($avID > 0)
    {
        $av = ObjectAttributeValue::getByID($avID);
        if (is_object($av))
        {
            $av->setObject($this);
            $av->setAttributeKey($ak);
        }
    }
    if ($createIfNotFound)
    {
        $cnt = 0;
        if (is_object($av))
        {
            $cnt = $db->GetOne("select count(avID) from
                ObjectAttributeValues where avID = ?",
                $av->getAttributeValueID());
        }
        if ((!is_object($av)) || ($cnt > 1))
        {
            $av = $ak->addAttributeValue();
        }
    }
    return $av;
}
```

This brings us to the end of the discussion about attribute keys. We will now focus on attribute types.

The attribute type

As already mentioned, attribute types are used to determine the nature of data that a given attribute key object can store. Attribute types can be simple and can store just bits of text or numbers, as well as being complex or advanced, and store the entire selections and lists, and so on.

concrete5 comes with a good deal of built-in attribute types too, which can be found at `/models/attribute/types/` (this path is relative to the `/concrete` folder and/or to `/concrete/core` of your concrete5 installation). Some of these built-in attribute types are as follows:

- `address`
- `default`
- `rating`
- `text`
- `boolean`
- `image_file`
- `date_time`
- `select`
- `textarea`
- `number`

Let's now focus on the key components of an attribute type in concrete5.

The database file

As with any other entity in concrete5, attribute types have their own `db.xml` file that uses the ADODB AXMLS format, as discussed in the previous chapter.

We will return to the `db.xml` file when discussing custom attribute types in this chapter, but for now, it will suffice to say that this file contains the schema of the concerned attribute type.

The `db.xml` file can contain multiple tables, but the core attribute type table must contain a column named `avID`, which is an unsigned integer.

Here is a sample db.xml file for the date_time attribute that is built-in within concrete5:

```
<xml version="1.0">
<schema version="0.3">
  <table name="atDateTimeSettings">
    <field name="akID" type="I" size="10">
      <KEY/>
      <UNSIGNED />
    </field>
    <field name="akDateDisplayMode" type="C" size="255" />
  </table>
  <table name="atDateTime">
    <field name="avID" type="I" size="10">
      <KEY/>
      <UNSIGNED />
    </field>
    <field name="value" type="T">
      <DEFAULT value="0000-00-00 00:00:00"/>
    </field>
  </table>
</schema>
</xml>
```

The attribute type controller

The controller.php file extends the AttributeTypeController object. The attribute type controller deals with the storage, saving, searching, retrieval, and manipulation of all the data associated with a given attribute type. Broadly speaking, the most simple attribute types might contain nothing beyond the db.xml and controller.php files and other minor ones such as icon files, and so on.

As a general rule, this file consists of the following methods and definitions, though some of these methods and properties are optional:

- `$searchIndexFieldDefinition`: This is a string/array that can be used to determine the schema of the attribute type. As a string, it uses the attribute key handle as the column name in the table (with `ak_` as its prefix). However, as an array, it creates multiple columns in the form of `ak_handle_KEY`, where `KEY` refers to the key of the column in the associative array. For example:

```
protected $searchIndexFieldDefinition = array
(
    // array definitions come here
);
```

- `getValue()`: This method queries the attribute type table and returns the unformatted value.
- `form()`: This method is executed before the `form.php` file is rendered. If the `form.php` file does not exist, elements of the form can be printed from this method itself.
- `type_form()`: This method is executed before the `type_form.php` file is rendered. Also, if the `type_form.php` file does not exist, elements can be printed from the method itself. This method also allows developers to specify additional settings for the given attribute type.
- `searchForm($list)`: Each time a database is queried, this method is used to filter results on the basis of the current attribute type.
- `getDisplayValue()`: This method deals with the presentational aspect of things. It gets a value, and then presents it in a human-readable format.
- `getDisplaySanitizedValue()`: This method gets the current value and then looks for potentially harmful characters, thereby *sanitizing* the value.
- `search()`: As the name suggests, this method is run before the `search.php` file is rendered. If the `search.php` file does not exist, form elements can be printed directly from the method itself.
- `saveValue($value)`: This method saves the value of the attribute type's table, and then binds it to the `avID` column that is an unsigned integer.
- `saveForm($data)`: This method is run when a form (rendered either via `form()` or `form.php`) is submitted. It accepts an associated array as `$data`.
- `duplicateKey($newAttributeKey)`: This method is automatically invoked when an attribute key is duplicated.
- `deleteKey()`: This is run when an attribute key is deleted, allowing the attribute type to clean up.
- `validateForm($data)`: This method comes into play if a particular attribute needs to be validated before it can proceed. If the attribute validates and the data is complete, the method returns `true`.
- `deleteValue()`: This method removes the data from the attribute type's table if it matches `$this->getAttributeValueID()`.

Apart from `db.xml` and `controller.php`, an attribute type also contains certain other files of lesser importance, as follows:

- **The `icon.png` file**: This is a 16 x 16 PNG icon file that is displayed when the attribute type is enlisted in the concrete5 Dashboard.
- **The `form.php` file**: This file is displayed when there is an interaction with the attribute type. It is optional.

- **The `type_form.php` file:** This is displayed (in the dashboard only) when the attributes of the given type are added. It is optional and is generally present only if the given attribute type requires a special configuration.
- **The JavaScript and CSS files:** The `form.js` and `form.css` files, if present, are autoloaded when `form.php` is rendered. Similarly, `type_form.js` and `type_form.css`, if present, are autoloaded when `type_form.php` is rendered.

So far, we have discussed and understood the major components of attribute categories, keys, and types. We now have an understanding of the major methods and properties associated with each, and have also discussed the basics of concrete5 attributes.

We shall now move on to the custom attribute types. Basically, you can create your own custom attribute type by simply copying one of the existing attributes, duplicating it, rewriting and editing parts of its controller classes and schema, and then installing the new custom attribute type via the concrete5 Dashboard. Let's discuss this procedure in detail.

Creating a custom attribute type in concrete5

As you have already read in this chapter, concrete5 comes with several attribute types of both simple and complex nature. For all practical purposes, the default attribute types tend to suffice for concrete5 development.

However, just in case you wish to extend the existing attribute types or create new custom attribute types, I will describe the steps to do so here.

Custom attribute types are generally employed only in really advanced practical or real-world solutions, especially in cases when the default attribute types provided by concrete5 are either too complex or too simple for our given usage scenario. For example, take the case of the e-commerce add-on of concrete5. If you are building an e-commerce website using concrete5, there is a good chance that you will need your buyer's address for shipping, billing, and so on. Thus, you can easily use the e-commerce add-on's address attribute types, such as `Billing_Address`, `Shipping_Address`, and so on.

But what if you do not wish to have a complicated address field? What if your buyers are all local, so that you just need a simple text area and possibly a custom list of cities? Of course, you can tweak the e-commerce add-on itself, but an easier and more sensible method will be to create a new custom attribute type for the address field to suit the validation requirements and other needs of your project. In terms of API, this custom attribute type will remain backward compatible, so you will have no difficulty in getting it to work with the e-commerce add-on of concrete5.

The following tutorial will teach you how to create a custom attribute type based on the address attribute, and name it `packt_address`:

1. First, we need to set up the directory structure in order. Navigate to the `/models/` folder within the root of the concrete5 installation that you are working on. Therein, create an `attribute` directory, and inside the `/models/attribute/` folder, add another subfolder named `types` so that our directory structure looks as follows:

```
<path-to-concrete5-root-folder>/models/attribute/types
```

This will serve as our local attribute types' directory for the given installation of concrete5.

2. Now, since our tutorial intends to create a custom attribute type on the basis of the address attribute type, we need to copy the address attribute type to the local attribute types' directory that we just created. At this point, let's rename the address attribute type directory to `packt_address`. This is how the directory structure will look, relative to the root of your concrete5 installation:

```
/models/attribute/types/packt_address
```

Also, it is a good practice to eliminate the items and features that your custom attribute type does not need or will not require. For example, since our sole purpose here is to simplify the address attribute type to suit the needs of our local e-commerce shops, it might make sense to eliminate fields that allow countries and provinces to be selected from the address section, or files and filters that automatically change provinces when the country is changed.

For the address attribute type, the following files need to be removed:

- `country_state.js`
- `type_form.php`
- `type_form.js`

Of course, this varies from one scenario to another, so you need to figure out and decide which files of the given concrete5 attribute type are not required or are vital for the custom attribute type that you are building, and then act accordingly.

The database file

Having gone through the previous chapters of this book, you should now be very well-acquainted with the role, importance, and functioning of the `db.xml` file in concrete5. As has already been discussed, the `db.xml` file is an XML representation of the concerned database tables that uses the AXMLS file format and is very popular in concrete5 development.

This file is automatically parsed and executed when the entity in question is called, which in our case will be the attribute type. Since we are working with a custom attribute type, we will need a custom attribute table to store the required information.

Here is how a sample `db.xml` for our custom attribute type will look:

```
<?xml version="1.0"?>
<schema version="0.3">
<table name="atPacktAddress">
    <field name="avID" type="I" size="10">
        <KEY></KEY>
        <DEFAULT value="0"></DEFAULT>
        <UNSIGNED></UNSIGNED>
    </field>
    <field name="address" type="X" ></field>
    <field name="city" type="C" size="300"></field>
</table>
</schema>
```

The preceding table is self-explanatory. We are just using one address field for the text type and another one for the city. This `db.xml` file, as is obvious, will be placed in the attribute type directory.

The form.php file

The `form.php` file comes into play each time the attribute type is shown. In our case, each time the user inputs his/her address, the `form.php` file will be invoked.

Open the existing `form.php` file from the address attribute type and modify it as follows to suit the needs of our custom attribute type:

```
<? defined('C5_EXECUTE') or die(_("Access Denied.")); ?>
<? $f = Loader::helper('form'); ?>
<div class="ccm-attribute-address-line">
```

```

<?=$f->label($this->field('address'), t('Address'))?>
<?=$f->textarea($this->field('address'), $address)?>
</div>
<div class="ccm-attribute-address-line">
<?=$f->label($this->field('city'), t('City'))?>
<?=$f->select($this->field('city'), array(
    'city1' => 'City 1',
    'city2' => 'City 2',
    'city3' => 'City 3'
), $city);
?>
</div>

```

In the preceding code, the users are being asked to enter their address, and then choose their city from the given list of cities.

The controller.php file

In the previous chapters, much like the `db.xml` file, you also learned about the `controller.php` file and the controller class as well as its usage and importance in concrete5. For our custom attribute type, we basically need to start with renaming the existing classes and then converting and modifying the existing methods to suit our purpose.

Look for the names of the controller classes, which will look something like this:

```

class AddressAttributeTypeController extends AttributeTypeController
{
....
class AddressAttributeTypeValue extends Object
{

```

Rename the preceding classes, shown as follows:

```

class PacktAddressAttributeTypeController extends
AddressAttributeTypeController
{
....
class PacktAddressAttributeTypeValue extends AddressAttributeTypeValue
{

```

Now it's time to tweak and modify the existing methods of the `controller.php` file. We will proceed with one method at a time, and modify each method as per our needs to properly convert the address attribute type into a custom attribute type suited for our website's requirements. The methods are discussed as follows:

- `searchKeywords`: This method is used when keywords are searched against the given attribute type. This is how it will look:

```
public function searchKeywords($keywords)
{
    $db = Loader::db();
    $qkeywords = $db->quote('%' . $keywords . '%');
    $str = '(ak_' . $this->attributeKey-
        >getAttributeKeyHandle() . '_address1 like
        '.$qkeywords.' or ' ;
    $str .= 'ak_' . $this->attributeKey-
        >getAttributeKeyHandle() . '_address2 like
        '.$qkeywords.' or ' ;
    $str .= 'ak_' . $this->attributeKey-
        >getAttributeKeyHandle() . '_city like
        '.$qkeywords.' or ' ;
    $str .= 'ak_' . $this->attributeKey-
        >getAttributeKeyHandle() . '_state_province
        like '.$qkeywords.' or ' ;
    $str .= 'ak_' . $this->attributeKey-
        >getAttributeKeyHandle() . '_postal_code
        like '.$qkeywords.' or ' ;
    $str .= 'ak_' . $this->attributeKey-
        >getAttributeKeyHandle() . '_country
        like '.$qkeywords.' )';
    return $str;
}
```

Modify it as follows:

```
public function searchKeywords($keywords)
{
    $db = Loader::db();
    $qkeywords = $db->quote('%' . $keywords . '%');
    $str = '(ak_' . $this->attributeKey-
        >getAttributeKeyHandle() . '_address
        like '.$qkeywords.' or ' ;
    $str .= 'ak_' . $this->attributeKey-
        >getAttributeKeyHandle() . '_city
        like '.$qkeywords.' or ' ;
    return $str;
}
```

Thus, we have removed the extra fields and are now only searching for address and city.

- **searchForm:** The `searchForm` method is called each time the advanced search form is used. Once again, since we do not need the excess functionality, we can safely replace the code with the following simplified version:

```
public function searchForm($list)
{
    $address = $this->request('address');
    $city = $this->request('city');
    if ($address)
    {
        $list->filterByAttribute(array('address' =>
            $this->attributeKey->getAttributeKeyHandle()),
            '%' . $address . '%', 'like');
    }
    if ($city)
    {
        $list->filterByAttribute(array('city' =>
            $this->attributeKey->getAttributeKeyHandle()),
            '%' . $city . '%', 'like');
    }
    return $list;
}
```

- **searchIndexFieldDefinition:** In this method, we just need to edit the array so that the correct columns and types are used by concrete5. Modify the array definition, as follows:

```
protected $searchIndexFieldDefinition = array(
    'address' => 'X NULL',
    'city' => 'C 300 NULL',
);
```

- **search:** The search method will require just a few minor tweaks, shown as follows:

```
public function search()
{
    print $this->form();
    $v = $this->getView();
    $this->set('search', true);
    $v->render('form');
}
```

- **validateForm:** The `validateForm` method will be run when the form is validated. We change it to reflect our newer and simpler address fields:

```
public function validateForm($data)
{
    return ($data['address'] != '' && $data['city'] != '');
}
```
- **saveForm:** The `saveForm` function remains unchanged, so this is how it should look:

```
public function saveForm($data)
{
    $this->saveValue($data);
}
```

- **getSearchIndexValue:** Once again, we modify this method simply to reflect our newer and small column size:

```
public function getSearchIndexValue()
{
    $v = $this->getValue();
    $args = array();
    $args['address'] = $v->getAddress();
    $args['city'] = $v->getCity();
    return $args;
}
```

- **saveValue:** Along similar lines as `getSearchIndexValue`, the `saveValue` method needs to be edited too in order to refer to the new database table and a small column set:

```
public function saveValue($data)
{
    $db = Loader::db();
    if ($data instanceof PacketAddressAttributeTypeValue)
    {
        $data = (array) $data;
    }
    extract($data);
    $db->Replace('atPacketAddress', array('avID' => $this->getAttributeValueID(),
    'address' => $address,
    'city' => $city
    ),
    'avID', true
    );
}
```

- `deleteKey`: The `deleteKey` method is called each time an attribute is removed. We need to modify it so that it reflects our new database table:

```
public function deleteKey()
{
    $db = Loader::db();
    $arr = $this->attributeKey->getAttributeValueIDList();
    foreach($arr as $id)
    {
        $db->Execute('delete from atPcktAddress
                    where avID = ?', array($id));
    }
}
```

- `deleteValue`: The `deleteValue` method is called each time an attribute's values are removed. Much like the `deleteKey` method, we need to modify the `deleteValue` method too so that it refers to the new database table:

```
public function deleteValue()
{
    $db = Loader::db();
    $db->Execute('delete from atPcktAddress
                where avID = ?', array($this-
                    >getAttributeValueID()));
}
```

- `getValue`: We just reflect the `PcktAddressAttributeTypeValue` class in this method:

```
public function getValue()
{
    $val = PcktAddressAttributeTypeValue::getByID
        ($this->getAttributeValueID());
    return $val;
}
```

- `getDisplayValue`: This method remains unchanged:

```
public function getDisplayValue()
{
    $v = $this->getValue();
    $ret = nl2br($v);
    return $ret;
}
```


- `load()`: This is how the load function will look:

```
public function form()
{
    if (is_object($this->attributeValue))
    {
        $value = $this->getAttributeValue()->getValue();
        $this->set('address', $value->getAddress());
        $this->set('city', $value->getCity());
    }
}
```

In the controller class of the address attribute type, you might find additional methods, such as:

- `action_load_provinces_js()`
- `validateKey`
- `duplicateKey`
- `saveKey`
- `type_form`

These methods are of no use for our custom attribute type, so you can safely eliminate them.

The **PacktAddressAttributeTypeValue** class

The `PacktAddressAttributeTypeValue` class is supposed to extend the `AddressAttributeTypeValue` class. Following the changes, this is how it is supposed to look:

```
public static function getByID($avID)
{
    $db = Loader::db();
    $value = $db->GetRow("select avID, address,
        city from atPacktAddress where avID = ?", array($avID));
    $aa = new PacktAddressAttributeTypeValue();
    $aa->setPropertiesFromArray($value);
    if ($value['avID'])
    {
        return $aa;
    }
}
public function getAddress() {return $this->address;}
```

```

public function getCity() {return $this->city;}

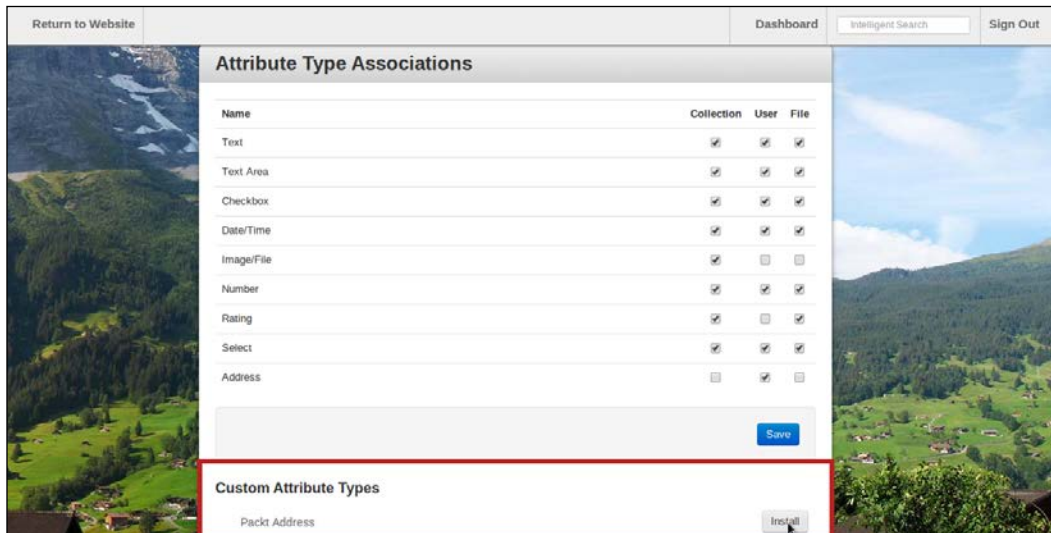
public function __toString()
{
    $ret = '';
    if ($this->address)
    {
        $ret .= $this->address . "\n";
    }
    if ($this->city)
    {
        $ret .= $this->city;
    }
    return $ret;
}

```

Installation of the custom attribute type

Once you are done building, tweaking, and modifying the custom attribute type, it is ready for installation on the concrete5 website. Since our custom attribute type intends to work for an e-commerce site, the e-commerce add-on and concrete5 needs to be informed about this attribute type.

In the concrete5 backend, navigate to **Dashboard | Settings | Attributes | Types**, and then, under **Attribute Type Associations**, look for the section titled **Custom Attribute Types**, as shown in the following screenshot:



You will see the **Packt_Address** type listed therein. Install it, and then make it available for all the required attribute categories. In our example, the e-commerce `Order` and `Users` categories will need to have access to this new custom attribute type.

Using the new custom attribute type

Lastly, we need to change address types in order to use the new custom attribute type. For e-commerce websites, the easiest way to do so is to head to the `Order` attributes and `User` attributes section, and then delete any other address type that might be present there.

Thereafter, you can add the address types again, albeit this time they will be using the new custom attribute type. Following that, each time the address types are required, the new custom attribute type will be used.

That brings us to the end of this tutorial about creation and modification of custom attribute types. You can modify aspects of this tutorial to suit your needs and work with custom attribute types for add-ons and solutions other than e-commerce as well.

Summary

With this, we come to the end of this chapter. In this penultimate chapter of the book, we have learned about attributes in concrete5. We discussed what attributes, their uses, and their roles in concrete5 are; then, we focused on attribute categories, keys, and types. Following this, we also understood the role of custom attribute types, and learned how to extend an existing attribute type and create a custom attribute type, using it to suit the requirements of our projects.

In the last chapter of the book, we will be focusing on the concrete5 permissioning system. Apart from basic and complex permissions, we will also learn how to manage permissions programmatically, and then move on to workflows in concrete5.

5

Permissions and Workflows

In the previous chapter, you learned about concrete5 attributes. We discussed details such as attribute keys, attribute categories, attribute types, and also mastered the creation of custom attribute types by means of extending a pre-existing attribute type.

In the final chapter of this book, our focus will shift toward permissions and workflows in concrete5. We will be taking a look at how to manage permissions in concrete5, simple and advanced permissions, programmatically dealing with permissions, and learning about the workflows of concrete5 and their configuration.

Here is a rundown of what this chapter will talk about:

- Permissions in concrete5
- Simple and advanced permissions
- Advanced permissions: enabling, editing, copying, working with, and manipulating area/block/file permissions
- Managing permissions programmatically
- The workflows of concrete5
- Basic workflow setup in concrete5

Most of this chapter will make use of screenshots from the concrete5 backend, though we will also be working with some levels of code every now and then.

Introducing permissions in concrete5

The permission system of concrete5 is very powerful and is used to control and decide which user gets to edit what.

While files in concrete5 have their own permission model, in this chapter, our focus will be on site wide permissions. Basically, concrete5 works with two modes of permissions as follows:

- **Simple permissions:** This is the default permission model. It lets you curtail the ability to edit and view the page and site data on the basis of user groups that can be configured as per your needs.
- **Advanced permissions:** As the name suggests, advanced permissions lets you control different roles at the page and block levels.

If you wish to switch from simple permissions to advanced permissions, the method is simple. You simply append the following line to your `site.php` file located in the `/config/` folder:

```
define('PERMISSIONS_MODEL', 'advanced');
```

Alternatively, you can just go to the **System and Settings** section of your concrete5 Dashboard and then select **Advanced Permissions** from the **Permissions** section and turn them on. Note that this option is easier and more preferable than the first one.

Be warned though, once you have enabled advanced permissions, there is no easy way to go back to simple permissions (actually, there is hardly any way to do so, excluding a possible restore from a database backup, which may or may not work). Do this only if you are sure that your website requires an advanced level of permission management.

The difference between simple and advanced permissions

So, how do simple permissions of concrete5 differ from advanced ones?

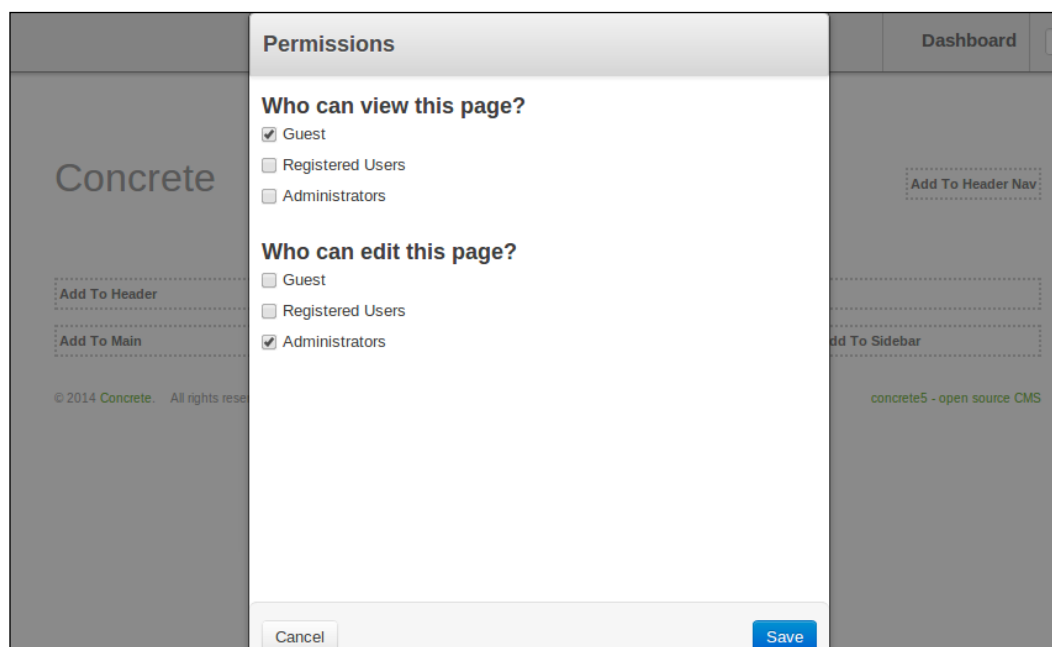
Basically, once you install concrete5, you are provided with simple permissions at your service. These permissions provide you with control over basic editing and access-related aspects of your website.

Advanced permissions, on the other hand, provide you with a finer control over your website. You can control the actions and access rules of users and groups in a flexible manner.

Simple permissions

Working with simple permissions in concrete5 is fairly easy. Once you put a page in the **Edit** mode, you will see a **Permissions** tab appear therein. Clicking on the tab gives you the permissions options for the concerned page.

Here is how the simple permissions option's section will look:



The screenshot shows the 'Permissions' dialog box in concrete5. The dialog has a title bar 'Permissions' and a 'Dashboard' button in the top right. The main content area is divided into two sections: 'Who can view this page?' and 'Who can edit this page?'. Each section has three checkboxes: 'Guest', 'Registered Users', and 'Administrators'. In the 'view' section, 'Guest' is checked. In the 'edit' section, 'Administrators' is checked. At the bottom, there are 'Cancel' and 'Save' buttons. The background shows a blurred view of the concrete5 interface with the 'Concrete' logo and some navigation links.

All your user groups are listed here, so you can edit and tweak the permissions for each user group as per your needs. If you add a new user group, it will automatically appear in the simple permissions option's section.

Pages in concrete5 tend to inherit permissions as they move down the tree. Thus, if I were to create a group for Sports Editors on my website and allow them to edit the Sports Updates page, they will also get the rights to edit any sports-related news that they might create on that page.

Advanced permissions

As already mentioned, the advanced permissions model of concrete5 allows you to have a greater control over your website. Advanced permissions provide you with control over features, such as page-specific access, block-specific control, and area-specific control. Beyond that, it also gives you some level of version control over pages and lets you add specific page types to specific sections of your website, if you desire to.

After enabling advanced permissions for your website, this is how the permissions section will look:

The screenshot shows the 'Permissions' section of the Concrete5 administration interface. The interface is divided into three main vertical panels. The left panel is a dark sidebar with the 'Concrete' logo and links for 'Add To Header', 'Add To Main', and a copyright notice for 2014. The middle panel, titled 'Permissions', contains two dropdown menus at the top: 'Assign Permissions' set to 'Manually' and 'Subpage Permissions' set to 'Inherit the permissions of this page.'. Below these is a list of permissions, each with a blue link and a grey button labeled 'Administrators'. The permissions listed are: View (with a 'Guest' button), View Versions, View Page in Sitemap, Preview Page As User, Edit Properties, Edit Contents, Edit Speed Settings, Change Theme, Change Page Type, Edit Permissions, Delete, Delete Versions, Approve Changes, and Add Sub-Page. At the bottom of this list are 'Cancel' and 'Save' buttons. The right panel is a dark sidebar with a 'Dashboard' link at the top and links for 'Add To Header Nav' and 'Add To Sidebar' further down.

Permission	Assigned To
View	Guest
View Versions	Administrators
View Page in Sitemap	Administrators
Preview Page As User	Administrators
Edit Properties	Administrators
Edit Contents	Administrators
Edit Speed Settings	Administrators
Change Theme	Administrators
Change Page Type	Administrators
Edit Permissions	Administrators
Delete	Administrators
Delete Versions	Administrators
Approve Changes	Administrators
Add Sub-Page	Administrators

A closer look at advanced permissions in concrete5

We will now discuss the advanced permissions of concrete5 in detail. Basically, we will talk about enabling advanced permissions, assigning such permissions, and some other related tasks.

Enabling advanced permissions

We have already discussed how to enable advanced permissions in concrete5. While doing so, there are some things that you need to bear in mind.

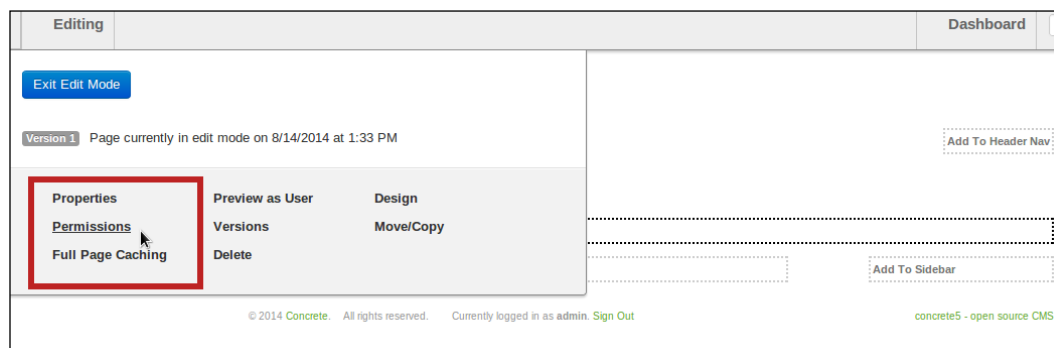
Firstly, the shift from simple to advanced permissions is permanent, unless you retain a database backup and attempt a restore. Enabling advanced permissions makes big changes to your database, and reverting back might not be always possible.

Next, the special `admin` superuser account that is created when concrete5 is installed always has unrestricted access to your website, irrespective of how you set up permissions. Thus, if you ever break something or need to override a setting, the `admin` user account is the way to go!

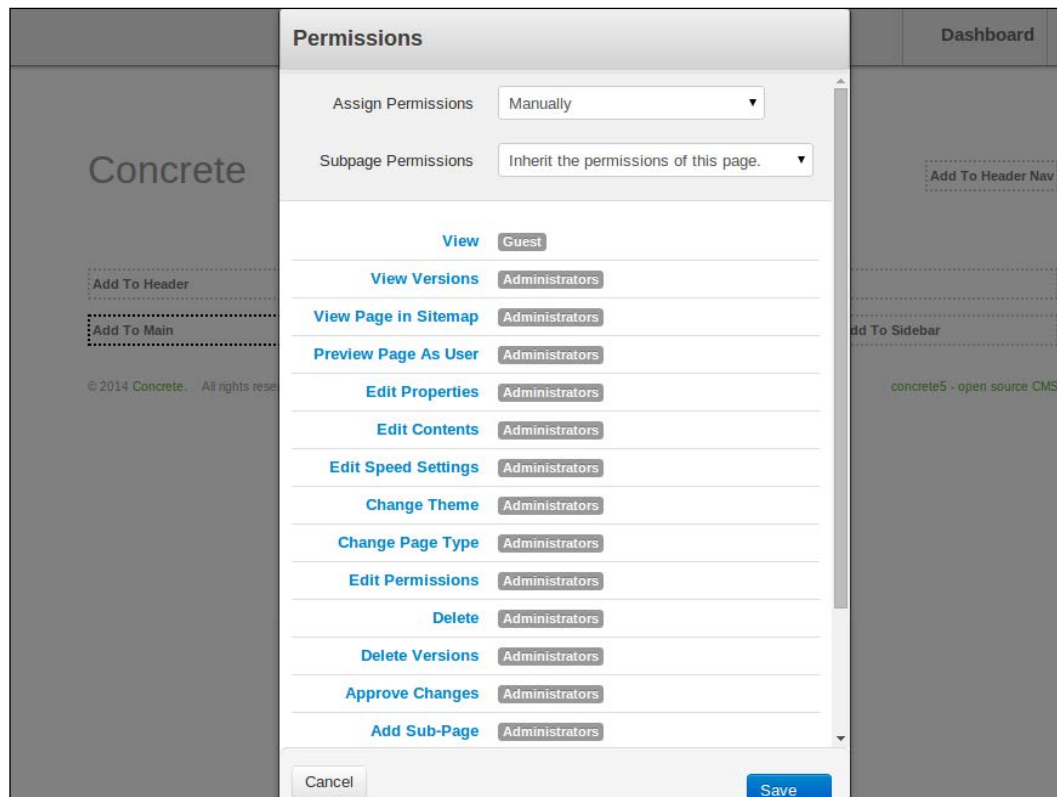
Assigning permissions

Assigning permissions in concrete5 is very easy. Note that you can also assign permissions via the site map page, if you desire to. The steps to assign permissions are explained in detail as follows:

1. Navigate to the page you want to assign permissions to and then switch to the **Edit** mode.
2. Thereafter, you will find the **Permissions** link, as shown in the following screenshot:



3. Once you click on the **Permissions** link, you will be presented with the **Permissions** dialog. It looks something like this:



Here is a rundown of the options and features of the **Permissions** dialog, as we can see in the preceding screenshot:

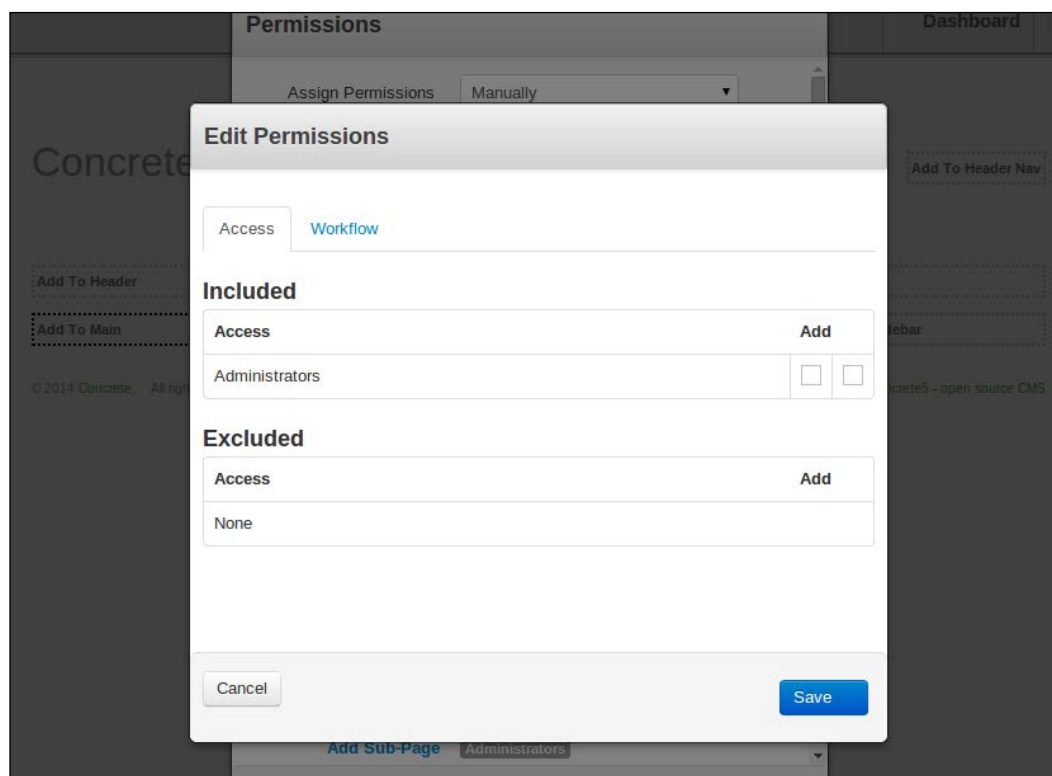
- The **Assign Permissions** section is of three types: **By Area of Site (Hierarchy)** sets the page to inherit the permissions that are set for its parent page. **From Page Type Defaults** applies the permissions, as they are set for the particular page type's default settings. **Manually** allows you to manually select permissions for the given page.
- Each of the individual items that we can set the permissions for are blue hyperlinks. Once you click on any of them, you can choose the permissions as per your needs.
- There is a second section too that mostly deals with the subpage permissions. **Inherit page type default permissions** sets the subpages of the current page to inherit the default permissions; whereas **Inherit the permissions of this page** sets the subpages to inherit the permissions of the current page.

If you wish to edit permissions, you should consider setting the permissions' assignment to the **Manual** mode.

Inclusion and exclusion

The users or groups that we wish to apply permissions for are known as **Access Entities**. Once you click on a permission object's link, you will see a list of Access Entities that are associated with the given permission type.

An Access Entity can either be included or excluded; if included, it has the right to perform the concerned action, and if excluded, it does not have the right to perform the said action, as you can see in the following screenshot:



You should bear in mind that exclusion will always override inclusion. Thus, if group A is included but a member of that group is explicitly excluded, then that particular member will not enjoy access that will otherwise be offered to other members of the group.

Timed permissions

You can use concrete5's time settings to make certain permissions valid only for a given time period. For instance, if you are running a news site and have a section that generally offers content only during the evening, you might want the concerned users or groups to have the required permissions for that section or page only toward the evening part of the day.

To set timed permissions, simply click the clock icon next to the access entity and then specify the time period as per your choice, shown as follows:

Add Access Entity

Access
Who gets access to this permission?

Administrators

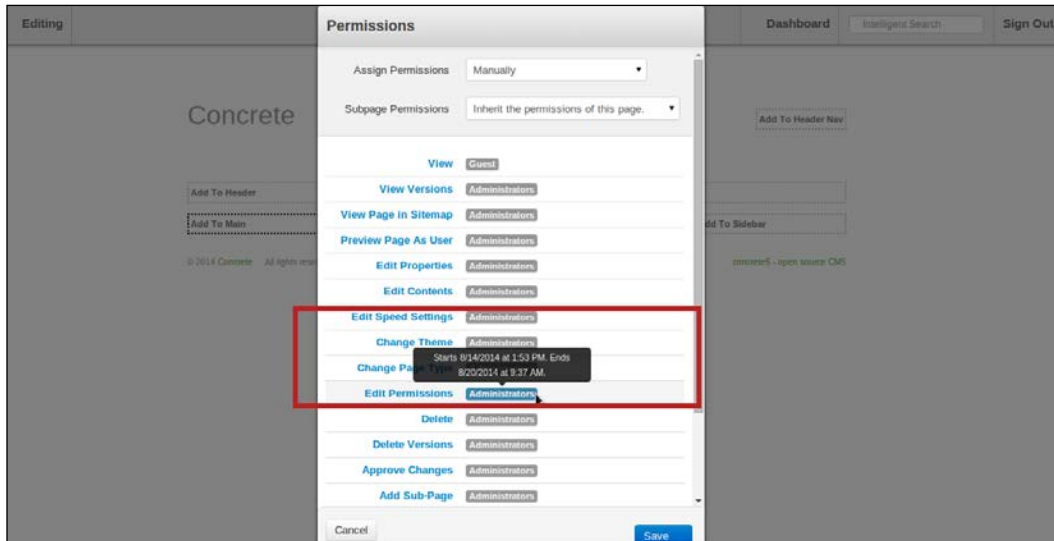
Time Settings
How long will this permission be valid for?

From ☒ 8/14/2014 1 : 53 PM ☐ All Day

To ☒ 8/20/2014 9 : 37 AM ☐ All Day

Cancel Save

Thereafter, that particular access entity will be highlighted and will show the time settings if you hover your cursor over it, shown as follows:

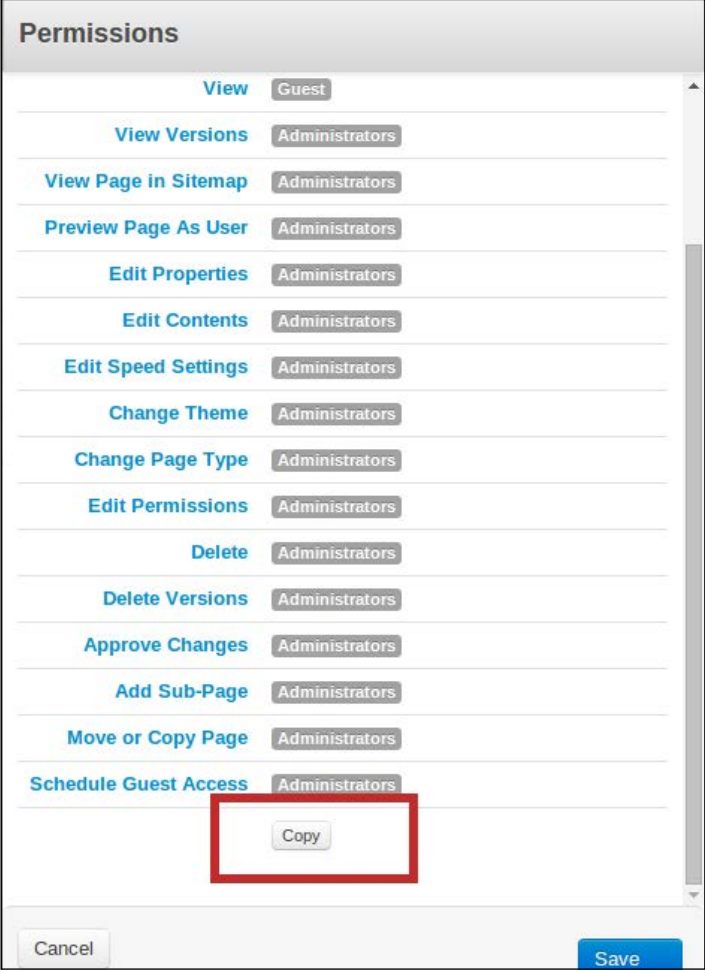


You can also set the timed permissions to repeat on a daily, weekly, or monthly basis.

Copying permissions

You can copy the permissions from one access entity to another simply by hovering over the concerned entity, dragging it to another item, and then releasing it.

If you wish to copy all the permission settings of a given page, use the **Copy** button present at the end of the list.



The screenshot shows a 'Permissions' dialog box with a list of permissions. The 'View' tab is selected, and the 'Guest' user is chosen. The list includes actions like 'View Versions', 'View Page in Sitemap', 'Preview Page As User', 'Edit Properties', 'Edit Contents', 'Edit Speed Settings', 'Change Theme', 'Change Page Type', 'Edit Permissions', 'Delete', 'Delete Versions', 'Approve Changes', 'Add Sub-Page', 'Move or Copy Page', and 'Schedule Guest Access'. Each action has a corresponding 'Administrators' button. At the bottom of the list, there is a 'Copy' button, which is highlighted with a red rectangular box. Below the list are 'Cancel' and 'Save' buttons.

Action	Permissions
View Versions	Administrators
View Page in Sitemap	Administrators
Preview Page As User	Administrators
Edit Properties	Administrators
Edit Contents	Administrators
Edit Speed Settings	Administrators
Change Theme	Administrators
Change Page Type	Administrators
Edit Permissions	Administrators
Delete	Administrators
Delete Versions	Administrators
Approve Changes	Administrators
Add Sub-Page	Administrators
Move or Copy Page	Administrators
Schedule Guest Access	Administrators
Copy	

Then, just navigate to the page that you wish to apply the permissions to, open its **Permissions** settings dialog and click on **Paste**. All the permissions from the source page will be copied to the destination page as well.

A note on area and block permissions

It makes sense to shed some light on area, block, and file permissions too.

By default, area and block permissions are inherited from the concerned page's permissions. You can, of course, override it by editing the permissions for the specific block or area.

For content areas on the page, you can set permissions by clicking on the concerned area and then selecting **Set Permissions** (generally the last option in the menu). The interface for doing this is the same as that of the page's permission settings, shown as follows:



Block permissions also follow more or less the same model and interface as that of area permissions, albeit there are differences in the sense that area permissions deal with viewing the area, adding block or stacks, and so on; whereas, block permissions address issues such as changing templates, viewing or editing the block, and so on. Block permissions also allow you to schedule guest access, which is basically specifying the time period during which that particular block can be accessed by guest users.

Lastly, speaking of file permissions, each time you upload a new file, it automatically inherits the global file permissions and the file permissions of the file set that the new file is added to. You can, of course, override it for individual files, if you desire to.

Managing advanced permissions programmatically in concrete5

As a developer or coder, you might wish to control your advanced permissions programmatically. For example, if a user adds a page, you may wish to back it up with additional information, describing the permissions system on your website.

There is hardly any sure shot *recommended* way of managing advanced permissions programmatically in concrete5. In fact, different coders use different routes to achieve their objectives.

The following code is my version of doing this. It was inspired by:

- Several failed and successful attempts of mine while working with advanced permissions and workflows in concrete5.
- The concrete5 documentation that has a run-through of how to programmatically set up advanced permissions. Find more info at <http://www.concrete5.org/documentation>.

Comments are placed wherever necessary to explain the functioning and logic behind the code:

```
//First, set up permissions of the page to make it visible to only the
relevant folks.
$pk = Page::getByPath('/some-nice-path');

//Manual Permission Mode activated!
$pk->setPermissionsToManualOverride();

//Child pages should inherit permissions.
$pk->setPermissionsInheritanceToOverride();

//Now changing view permissions.
$pk = PagePermissionKey::getByHandle("view_page");

//Setting up the current page.
$pk->setPermissionObject($p);

//Returning the access object.
$paGlobal = PermissionAccess::getByID($pk->getPermissionAccessID(),
$pk);

//Time to duplicate the access object.
```

```

$pa = $paGlobal->duplicate();

//Getting the groups that need to be added to the view_page permission
key.
$addSomeGroups = array(
    Group::getByName("One Grp Users"),
    Group::getByName("More Grp Users")
);

//Assigning the groups to the newly-created permission access object.
foreach ($addSomeGroups as $addPls)
{
    $pe = GroupPermissionAccessEntity::getOrCreate($addPls);
    $pa->addListItem($pe, false, 10);
}

//Getting the groups that need to be removed from the permissions.
$removeSomeGroups = array(
    Group::getByName("Grp of Ppl"),
    Group::getByName("Another Grp")
);

//Now removing them from the permissions access object.
foreach ($removeSomeGroups as $removePls)
{
    $pe =
GroupPermissionAccessEntity::getOrCreate(Group::getByName($removePls));
    $pa->removeListItem($pe);
}

//Saving the Permission Configuration.
$pa->save(array('paID' => $pa->getPermissionAccessID()));

//Getting the permission reference for the page.
$pt = $pk->getPermissionAssignmentObject();

//Providing it with the new configuration
$pt->assignPermissionAccess($pa);

//Finally, elevating the privileges.
$pa = GroupCombinationPermissionAccessEntity::getOrCreate(array_
merge($groups, array(Group::getByName('Grp Accounts'))));
//All done!

```

You can modify the preceding code to suit your needs, and once done with that, programmatically managed advanced permissions can reduce redundant tasks and help you save time in the long run.

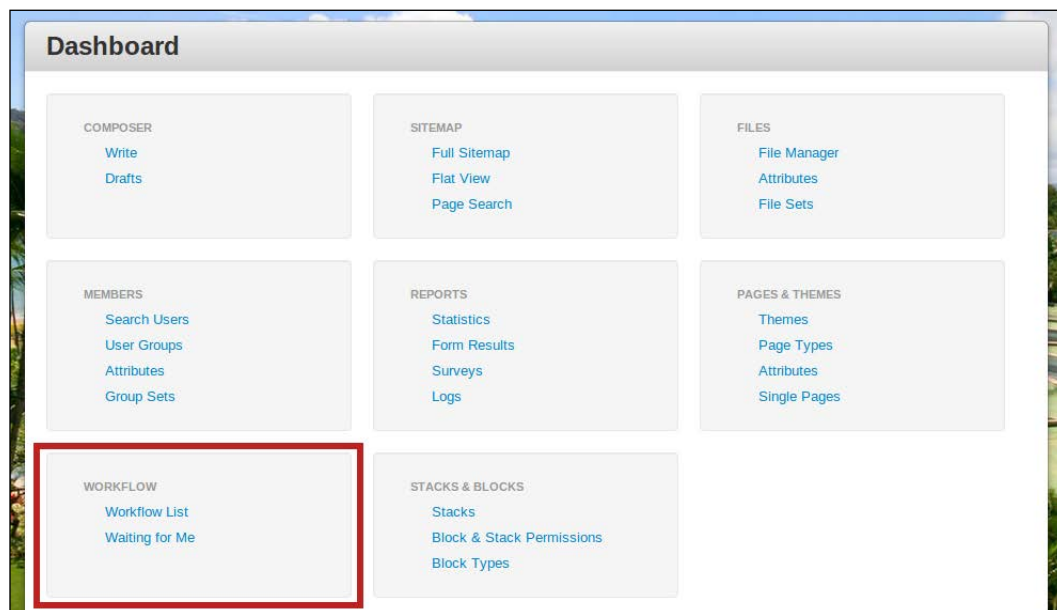
Workflows in concrete5

In concrete5, workflows are used to enhance and promote collaboration between users and groups for various tasks pertaining to content creation, be it editing or deleting. As such, workflows act as a buffer or an intermediate step between editing and publishing content. For example, you can use workflows to proofread or verify content before it goes live, and so on.

Workflows work in assonance with the enhanced level of control that is offered by advanced permissions in concrete5 and as such, you need to have the advanced permissions activated in order to set up and make use of workflows within concrete5.

Basically, there are two types of workflows in concrete5:

- **Basic workflow:** Basic workflow comes bundled with the concrete5 core. It is nothing more than a single-step approval measure that can be added to any item that supports advanced permissions on any given page. Basic workflow also offers notifications to help users and groups stay abreast with the changes as and when they are made. If you need to manage different permission items on a page or across pages, you can create multiple workflows as well. You can manage workflows by navigating to **Dashboard | WORKFLOW**, shown as follows:





We will return to Basic workflows of concrete5 later.

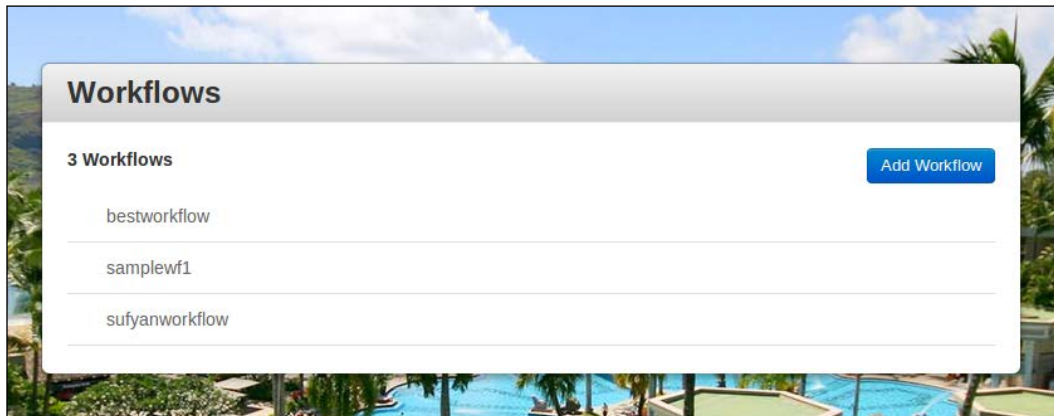
- **Enterprise workflow:** Primarily aimed at large organizations that work with multiple teams, Enterprise workflow of concrete5 seeks to extend the core workflow functionality by adding a multistep process to the workflow. In this method, changes are passed along various steps or a sequence of workflow measures, thereby allowing you to have a better control over the workflow management and the overall process. It can control page permissions in a stepwise manner, which lets you decide which users or groups can view or modify a page at a given step in the workflow.

More about the basic workflow

We shall now discuss the setup, configuration, and modus operandi of Basic workflow.

The workflow list

The workflow list displays all the workflows that you have set up on your concrete5 website. You can also add a new workflow by clicking on **Add Workflow**, shown as follows:



The Waiting for Me section

The **Waiting for Me** section enlists all the workflow items that are currently waiting for your review, approval, and/or denial.

This section is divided in three parts, thereby showing the relevant changes that have pending approval/disapproval of each part: Pages, Files, and Users.

Here is an overview of the details offered by the **Waiting for Me** section:

- **Name** lists the name of the page that is affected by the change.
- **URL** provides the URL to the said page.
- **Last Action** shows when the page was submitted for the administrator's review (in the single-step Basic workflow structure).
- **Current Status** shows the status for which the said page is pending a review. For example, if the user requests an approval for the page edit, it will show as **Pending Approval**.
- There are additional buttons right next to each entry to help you: **Approve**, **Delete**, **Cancel**, or **Compare Versions**. The **Compare Versions** feature, as expected, lets you load both the before-edit and after-edit versions of the page, so that you can decide for yourself whether the edits made are in proper form and order prior to approving or deleting them.

If you do not like the page in its current state, you can cancel the request, which will return the page to the user and allow them to resubmit it with further editing. Thereafter, you can approve the page from the **Waiting for Me** section itself, which will publish the changes and make them live on the website.

Setting up the basic workflow

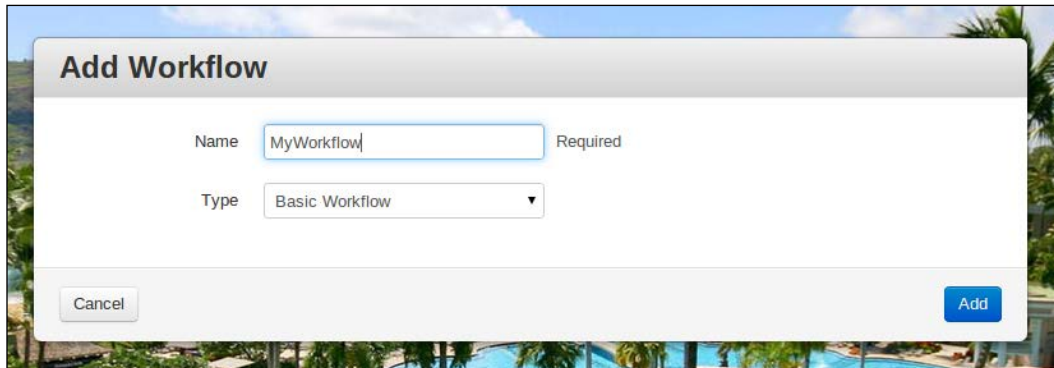
Now, we will focus on how to set up, create, and manage the basic workflow in concrete5.

Creating the workflow

Creating the workflow is simple, as follows:

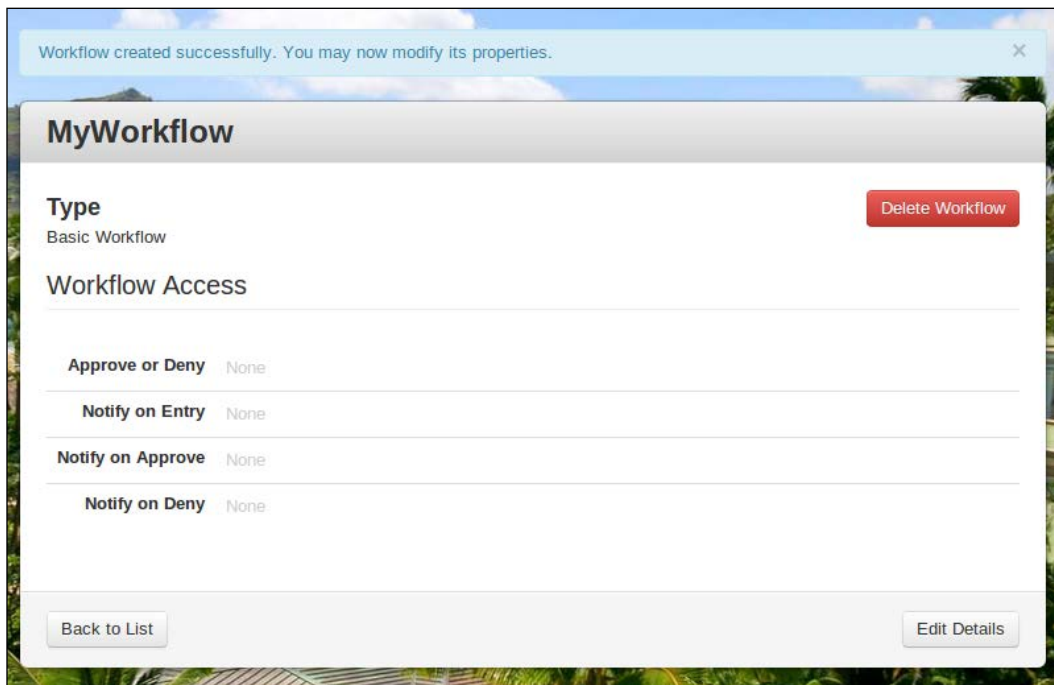
1. We need to navigate to **Dashboard | Workflow | Workflow List** and then select **Add Workflow**.

2. On the following screen, we will specify the name of the workflow and then click on **Add**:



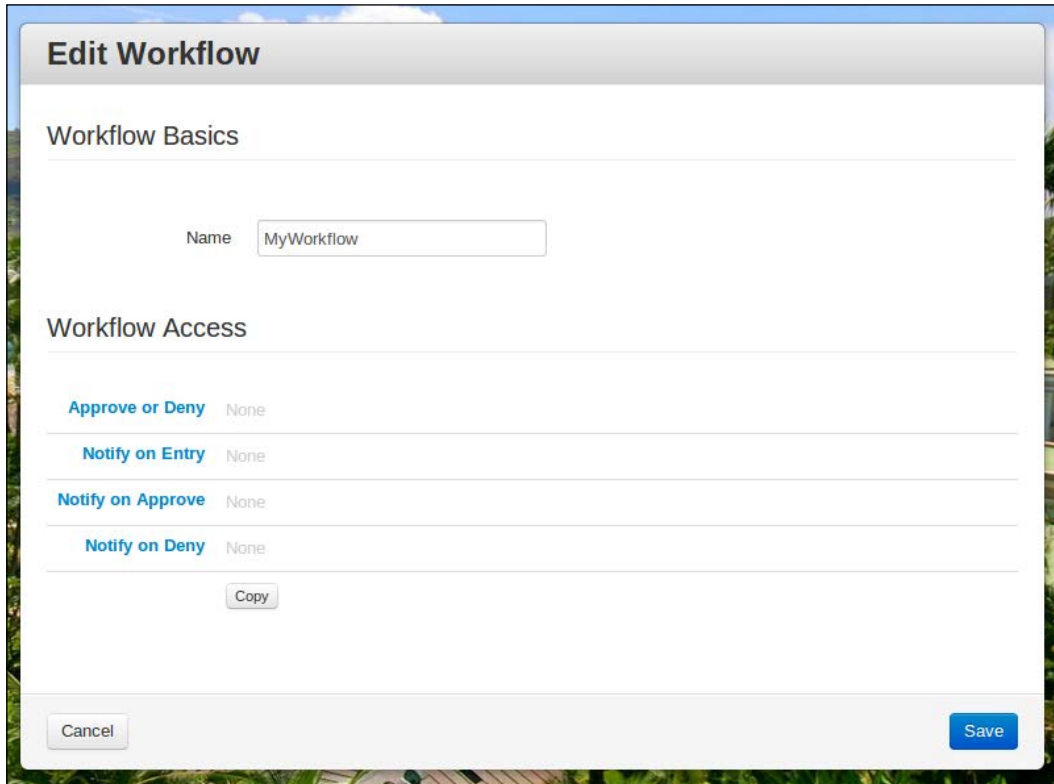
The screenshot shows a dialog box titled "Add Workflow". It contains two input fields: "Name" with the text "MyWorkflow" and a "Required" label, and "Type" with a dropdown menu showing "Basic Workflow". At the bottom, there are "Cancel" and "Add" buttons.

3. That's it. The workflow has been created; it is now time to modify its properties by hitting **Edit Details**.



The screenshot shows a page titled "MyWorkflow". At the top, a light blue banner reads "Workflow created successfully. You may now modify its properties." with a close button. Below the title, the "Type" is "Basic Workflow" and there is a "Delete Workflow" button. The "Workflow Access" section contains four rows, each with a label and a "None" value: "Approve or Deny", "Notify on Entry", "Notify on Approve", and "Notify on Deny". At the bottom, there are "Back to List" and "Edit Details" buttons.

4. In the editing section, you can specify who gets notified when (on entry, approval, or denial), as shown in the following screenshot:



The screenshot shows a dialog box titled "Edit Workflow". It has two main sections: "Workflow Basics" and "Workflow Access".

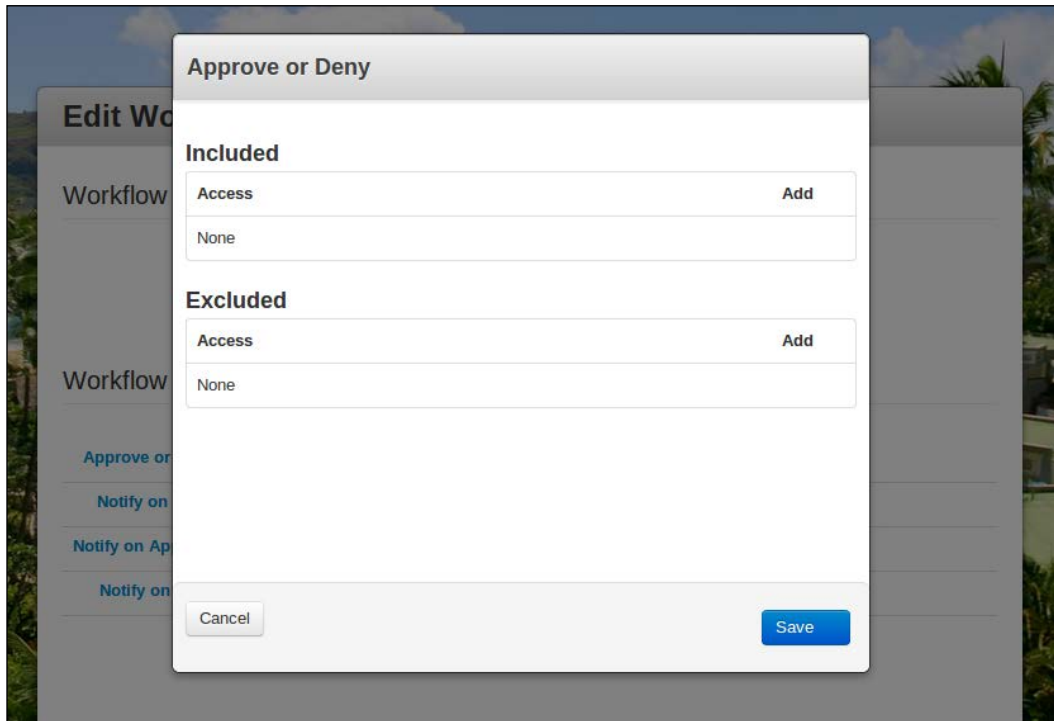
Workflow Basics

Name:

Workflow Access

Approve or Deny	None
Notify on Entry	None
Notify on Approve	None
Notify on Deny	None

5. You also have to specify who gets to approve or deny.



6. You can add multiple entities to each of the sections under **Workflow Access**, and once you are done, simply hit **Save** and your workflow will be modified with the changes made by you.

Attaching workflow to page permissions

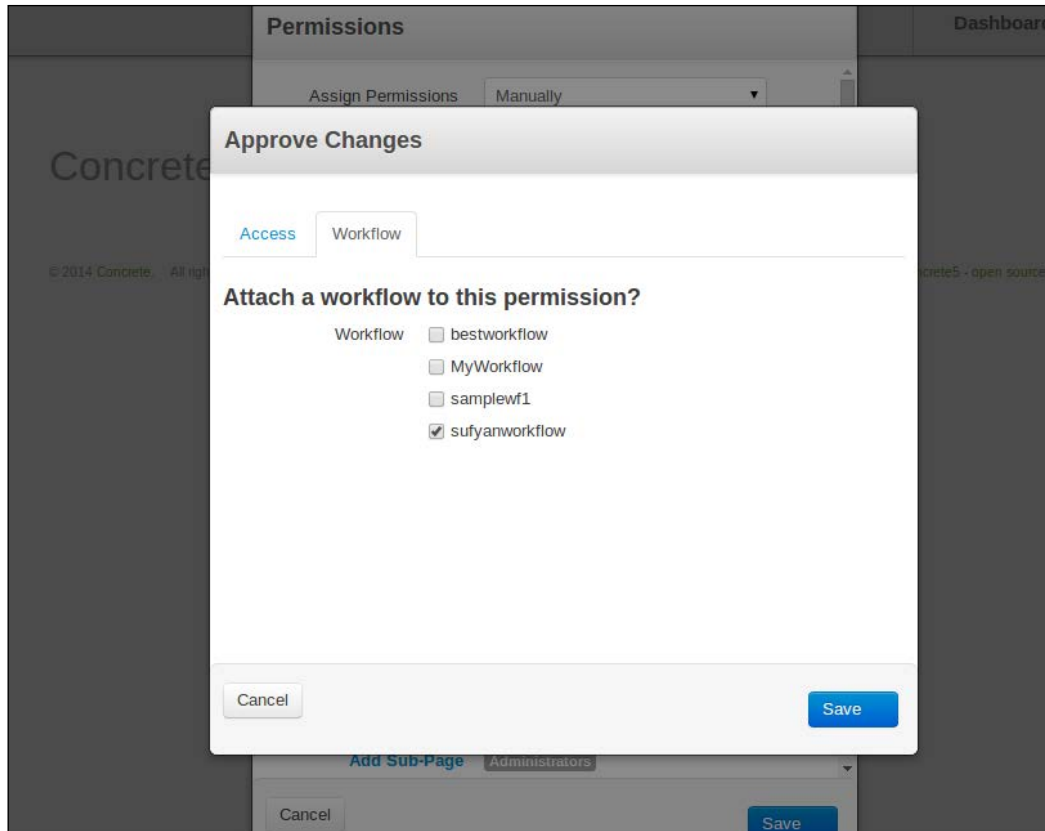
After creating and setting up your workflow, you need to attach it to a particular permission on a particular page. The steps are given as follows:

1. First, go to the concerned page, switch it to the **Edit** mode, and then select the **Permissions** option. Make sure that you have selected **Manually** under **Assign Permissions**, shown as follows:

The screenshot shows the 'Permissions' dialog box in Concrete5. The 'Assign Permissions' dropdown is set to 'Manually' and is highlighted with a red box. Below it, the 'Subpage Permissions' dropdown is set to 'Inherit the permissions of this page.' A list of permissions follows, each with a 'View' link and a user selection button (mostly 'Administrators'). At the bottom are 'Cancel' and 'Save' buttons.

Permission	User
View	Guest
View Versions	Administrators
View Page in Sitemap	Administrators
Preview Page As User	Administrators
Edit Properties	Administrators
Edit Contents	Administrators
Edit Speed Settings	Administrators
Change Theme	Administrators
Change Page Type	Administrators
Edit Permissions	Administrators
Delete	Administrators
Delete Versions	Administrators
Approve Changes	Administrators
Add Sub-Page	Administrators

2. Then, click on the function that you wish to assign workflow to (say, **Approve Changes**). In the panel that appears, switch to the **Workflow** tab. Select the workflow that you wish to attach to this particular permission type and hit **Save**, as shown in the following screenshot:



That's all! You have successfully attached your workflow to the concerned permission on the relevant page.

It must be noted though that certain page permissions do not have a **Workflow** tab present because they represent page properties that are managed by versions in concrete5. You can, however, attach the workflow to the **Approve Changes** permissions item for such page permissions.

As of the current release of concrete5, the following page permissions are managed by versions and do not have a **Workflow** tab attached to them:

- **Edit Content**
- **Edit Properties**
- **Edit Speed Settings**
- **Schedule Guest Access**
- **Add Sub-page**
- **Change Theme**
- **Change Page Type**
- **Preview Page as User**
- **View Versions**
- **View**

Modus operandi

Now that we have created the workflow, set it up, and even applied relevant page permissions to it, our task is done and we can start using it in the process of content creation.

I have already explained how the pending items that need your approval or denial are enlisted in the **Waiting for Me** section, and that is where most of the action happens.

Beyond that, concrete5 automatically sets notification e-mails to the concerned users as per the specifications provided by you. Therefore, your users can be notified for tasks that pertain to them. Here is how a sample notification e-mail will look:

Dear sufyan,

On August 10, 2014 at 2:04 AM, user jonny submitted the following request:

"Test Page" has been marked for deletion. View the page here:

<http://www.testrig.sufyanism.com/index.php/about/test-page/>.

You can review, approve or deny all pending actions from here:

<http://www.testrig.sufyanism.com/index.php/dashboard/workflow/me/>

Once again, the approval/denial and other review tasks happen from the **Waiting for Me** page of the concerned user's account in the concrete5 Dashboard. As already mentioned, the **Waiting for Me** page also gives information such as timestamps and other details, alongside the compare versions tool, thereby making the user's task fairly easy.

This brings us to the end of this discussion about workflows in concrete5. Once you have set up advanced permissions and created workflows, you can easily manage your website, especially if yours is a site with multiple users that need a workflow mechanism for better coordination.

Summary

This brings us to the end of the final chapter of this book.

In this chapter, you learned about permissions in concrete5. We talked about simple permissions, and then discovered the advanced permission model. Activating, using, and working with the advanced permission structure of concrete5 were discussed in this chapter, along with other relevant information such as timed permissions, programmatically dealing with advanced permissions, and so on.

Then, we also took a look at workflows in concrete5, and how to use them to our advantage. Tasks such as creating, setting up, and working with workflows were all covered in this chapter itself.

Thus, in this short book, you learned how to build concrete5 themes, work with basic and advanced blocks, manipulate and read data from the database, attributes and attribute types as well as the permissions model of concrete5. I hope you had a good time reading this book.

Happy coding!

Index

Symbols

`<body>` tag 14

`<head>` tag 13

A

Access Entities 77

ADODB AXMLS format 22

ADODB database extraction layer

benefits 37

ADODB PHP library

URL 40

advanced concrete5 blocks

composer 41

file management 42

working with 41

advanced permissions

about 72, 74

assigning 75-77

copying 80

enabling 75

exclusion 77

inclusion 77

managing 82, 83

timed permissions, setting 78, 79

versus simple permissions 72

area permissions 81

assets 23

attribute category

about 48

creating 50

File 49

Page 49

User 49

attribute key

about 49, 50

class file, creating 51

database file, creating 51

view page, creating for

attribute category 55

attributes 47, 48

attribute type

about 49, 57

address 57

boolean 57

CSS file 60

date_time 57

default 57

form.php file 59

icon.png file 59

image_file 57

JavaScript file 60

number 57

rating 57

select 57

text 57

textarea 57

type_form.php file 60

attribute type, components

attribute type controller 58-60

database file 57

B

basic workflow

about 84, 85

attaching, to page permissions 90, 91

creating 86-89

modus operandi 92

- setting up 86
- Waiting for Me section 86
- workflow list 85

block

- creating 27-29

block permissions 81

C

class file

- creating 51
- ObjctAttributeKey class 52, 53
- ObjctAttributeValue class 54

composer 41

concrete5

- attributes 47, 48
- block, creating 27-30
- code, packaging for distribution 44
- database tables, using 35, 36
- default functionality, overriding 42
- MVC framework 25
- packages 44
- permissions 72
- user inputs, validating 31-33
- workflows 84

concrete5 blocks

- about 21
- directory structure 22

concrete5 file manager, arguments

- \$args 42
- \$ff 42
- \$id 42
- \$postname 42
- \$text 42

concrete5 marketplace, rules

- browser testing 19
- changelog file, creating 19
- icon 19
- license 19

concrete5 theme

- building 5, 6
- default page type, creating 13
- development environment 7
- development prerequisites 7
- development tools 7
- Header Nav 6

- Main 6

- packaging, for marketplace 19

- page templates 6

- page types 5, 6

- Sidebar 6

- submitting, to marketplace 20

- templates, designing 8

concrete5 theme creation

- about 10
- description file, creating 11
- style sheet, editing 12, 13
- template, installing 10, 11
- thumbnail image, creating 11

contact_submit method 33

content areas

- about 14
- footer 18
- main or primary content 15
- miscellaneous 18
- navigation 17
- sidebar 16, 17

controller class

- \$btIncludeAll variable 25
- \$btInterfaceHeight variable 25
- \$btInterfaceWidth variable 25
- \$btTable variable 25
- add() function 25
- addHeaderItem(\$file) function 26
- delete() function 25
- duplicate(\$newBlockID) function 25
- edit() function 25
- getBlockTypeDescription() function 26
- getBlockTypeName() function 26
- getInterfaceHeight() function 26
- getInterfaceWidth() function 26
- getPermissionsObject() function 26
- on_page_view() function 25
- on_start() function 25
- render(\$view) function 26
- save(\$args) function 26
- set(\$key, \$value) function 26
- view() function 25

controller.php file

- \$searchIndexFieldDefinition property 58
- about 45, 58
- creating, for custom attribute type 63-67

- deleteKey() method 59, 67
- deleteValue() method 59, 67
- duplicateKey(\$newAttributeKey)
 - method 59
- form() method 59
- getDisplaySanitizedValue() method 59
- getDisplayValue() method 59, 67
- getSearchIndexValue() method 66
- getValue() method 59, 67
- load() method 68
- saveForm() method 59, 66
- saveValue() method 59, 66
- searchForm() method 59, 65
- searchIndexFieldDefinition() method 65
- searchKeywords() method 64
- search() method 59, 65
- type_form() method 59
- validateForm() method 59, 66

core templates

- add.php file 23
- edit.php file 23
- scrapbook.php file 23
- view.php file 23

custom attribute type

- controller.php file 63-68
- creating 60, 61
- database file 62
- form.php file 62
- installation 69, 70
- PacktAddressAttributeTypeValue class 68
- using 70

custom view templates 23

D

database

- data, obtaining with GetAll
 - command 39, 40
- data, obtaining with GetOne command 38
- data, obtaining with GetRow command 39
- debugging 41

database connections 36

database debugging mode 41

database file

- creating, for attribute key 51
- creating, for attribute type 57

- creating, for custom attribute type 62

database queries

- about 37
- using 37

database tables

- database connections 36
- database queries 37
- using 35, 36

db.xml file 22, 57

default page type

- <body> tag 14
- <head> tag 13
- creating 13
- editable content areas, building 14
- single page template 19

default.php template 6

development environment, concrete5 theme

- reference links 7

directory structure, concrete5 blocks

- about 22
- assets 23
- controller.php file 22
- core templates 23
- custom view templates 23
- db.xml file 22
- icon.png 22
- tools 24

E

enterprise workflow 85

Execute method 37

F

field function 42

file management 42

footer_required element 18

form.php file

- creating, for custom attribute type 62

full.php template 6

G

GetAll command

- used, for obtaining data
 - from database 39, 40

GetOne command
used, for obtaining data from database 38
GetRow command
used, for obtaining data from database 39
Global Area 17

I

icon.png 22
installation, custom attribute type
performing 69, 70
install() method 46

L

left_sidebar.php template 6

M

main.css style sheet
elements 12
modus operandi 92
MVC framework, concrete5 25

O

ObjectAttributeKey class
\$searchIndexFieldDefinition property 52
about 52
add(\$type, \$args, \$pkg = false) method 53
delete() method 53
getAttributes(\$primaryKey,
\$method = 'getValue') method 52
getAttributeValue(\$avID,
\$method = 'getValue') method 53
getByHandle(\$akHandle) method 53
getByID(\$akID) method 53
getColumnHeaderList() method 52
getIndexedSearchTable() method 52
getList() method 53
getSearchableIndexedList() method 52
getSearchableList() method 52
saveAttribute(\$object, \$value = false)
method 53
update(\$args) method 53

ObjectAttributeValue class
about 54
delete() method 54
getByID(\$avID) method 54
set* method 54
overrides
implementing 43
using 43

P

package controller
about 45, 46
considerations 46
packages, concrete5
about 44
package controller 45, 46
PacktAddressAttributeTypeValue class 68
page templates, concrete5 theme
default.php 6
full.php 6
left_sidebar.php 6
view.php 6
permissions, concrete5
about 72
advanced permissions 72
area permissions 81
block permissions 81
simple permissions 72

R

rss.php file 24

S

setBlockLimit(1) parameter 17
simple permissions
about 72, 73
versus advanced permissions 72
single page template
creating 19

T

templates

- designing 8
- design issues 10
- images 9, 10
- navigation menus 9
- websites, types 8, 9

timed permissions

- setting 78, 79

tools 24

typography.css

- properties 13

U

upgrade() method 46

user inputs

- validating 31-33

V

view page, attribute category

- attribute capabilities, adding to
 - objects 55, 56
- creating 55

view.php template 6

W

Waiting for Me section, basic workflow 86

workflow list 85

workflows

- about 84
- basic workflow 84, 85
- enterprise workflow 85

WYSIWYG editor 12