From Technologies to Solutions

Learning Joomla! 1.5 Extension Development

Creating Modules, Components, and Plug-Ins with PHP

A practical tutorial for creating your first Joomla! 1.5 extensions with PHP

Joseph LeBlanc

PACKT PUBLISHING

Learning Joomla! 1.5 Extension Development

Creating Modules, Components, and Plug-Ins with PHP

A practical tutorial for creating your first Joomla! 1.5 extensions with PHP

Joseph LeBlanc

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

Learning Joomla! 1.5 Extension Development

Creating Modules, Components, and Plug-Ins with PHP

First published: May 2007

Production Reference: 1180507

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

ISBN 978-1-847191-30-4

`www.packtpub.com`

Cover Image by `www.visionwt.com`

Credits

Author
Joseph LeBlanc

Reviewer
Riccardo Tacconi

Development Editor
Douglas Paterson

Assistant Development Editor
Mithil Kulkarni

Technical Editor
Akshara Aware

Editorial Manager
Dipali Chittar

Project Manager
Patricia Weir

Project Coordinator
Abhijeet Deobhakta

Indexer
Bhushan Pangaonkar

Proofreader
Chris Smith

Production Coordinator
Manjiri Nadkarni
Shantanu Zagade

Cover Designer
Manjiri Nadkarni

About the Author

Joseph LeBlanc started with computers at a very young age. His independent education gave him the flexibility to experiment and learn computer science. Joseph holds a bachelors degree in Management Information Systems from Oral Roberts University.

Joseph is currently a freelance Joomla! extension developer. He released a component tutorial in May 2004, which was later translated into French, Polish, and Russian. Work samples and open-source extensions are available at `www.jlleblanc.com`. In addition to freelancing, he served as a board member of the inaugural DC PHP Conference. He has also worked as a programmer for a web communications firm in Washington, DC.

I would like to thank the following people for making this book possible:

Packt Publishing, for giving me the opportunity to author this work.

The Joomla! Team, for developing some of the best software in the world.

Keith Casey, Conrad Decker, Jim Johnson, and Robyn Wyrick for their insights and advice on managing software companies.

To Steve and Sue Meeks, for their flexibility with my schedule during the writing process and for giving Joomla! a shot.

Everyone who has downloaded and used my open-source components.

My professors, for taking me on the Journey of a Byte and showing me how to write effectively.

Mom and Dad, for teaching me how to learn.

About the Reviewer

Riccardo Tacconi works for an Italian company as a system administrator and web developer using PHP, MySql, and Oracle. He is an MCP and studies IT part-time at the British Open University. His main interests are web development, Windows and Linux administration, Robotics, and Java software development (JMF, motion detection, CV and distributed systems).

He loves Linux and he is a proud member of the local Linux User Group: GROLUG. He tries to innovate ways to substitute Windows based technologies with Linux and open-source alternatives.

Table of Contents

Preface

Joomla! is an award-winning content management system with a powerful extension system. This makes it easy for third-party developers to build code extending Joomla's core functionality without hacking or modifying the core code.

Once an extension is developed, it can be packaged into a ZIP file for site administrators to upload and use. The people who manage Joomla!-based websites and want to use extensions need not know any programming at all. Once the ZIP file is uploaded, the extension is installed.

The name Joomla! comes from the Swahili word 'jumla', meaning "all together" or "as a whole". When you install an extension in Joomla!, it blends in with the rest of the site; all the extensions truly appear "all together, as a whole".

What This Book Covers

Chapter 1 gives an overview of how Joomla! works. The example project used throughout the book is also introduced. The three types of extensions (components, modules, and plug-ins) are covered along with descriptions of how they work together.

Chapter 2 begins the development of the component used in the project. Initial entries are made in the database and toolbars for the back end are built. The general file structure of Joomla! is also introduced.

Chapter 3 walks through the creation of the back-end interface for creating, editing, and deleting records in the project. Database table classes are introduced, as well as common HTML elements used to make the project blend in with other Joomla! extensions.

Chapter 4 builds a front-end interface for listing and viewing records. Additionally, code to generate and interpret search-engine-friendly links is covered. The project is also expanded slightly when a commenting feature is added.

Chapter 5 introduces a module used to list records on every page of the site. The module takes advantage of layouts, where the same data can be formatted differently depending on how the code is called. Some of the code is also separated out into a helper class so that the main code generating the module stays simple.

Chapter 6 rewrites the component developed in Chapters 2, 3, and 4 so that it follows the Model, View, Controller design pattern. Controls over the publishing of records are introduced, in addition to an interface for removing offensive comments. More toolbars are added and the search-engine-friendly URL code is redesigned.

Chapter 7 develops three plug-ins. The first plug-in finds the names of records in the database and turns them in to links to those records. A second plug-in displays a short summary of the record when certain code is added to content articles. Finally, another plug-in is designed so that records are pulled up along with Joomla! content searches.

Chapter 8 adds configuration parameters to the component, module, and plug-ins. These are handled through XML and generate a predictable interface in the back end for setting options. Retrieving the values of these parameters is standardized through built-in functions.

Chapter 9 expands the XML files used for parameters and adds a listing of all the files in each extension. Once this file is compressed along with the rest of the code into a ZIP archive, it is ready to be installed on another copy of Joomla! without any programmer intervention. Custom installation scripts and SQL code are also added to the component.

Code testing was performed using Joomla 1.5 beta 2.

Conventions

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

There are three styles for code. Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code will be set as follows:

```
function showReviews( $option )
{
  $query = "SELECT * FROM #__reviews";
```

```
      $db->setQuery( $query );
      $rows = $db->loadObjectList();
      if ($db->getErrorNum()) {
          echo $db->stderr();
          return false;
      }
    }
```

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

```
    switch($task)
    {
      case 'add':
      editReview( $option );
        break;
      case 'save':
      saveReview( $option );
        break;
    }
```

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

```
INSERT INTO jos_components (name, link, admin_menu_link,
            admin_menu_alt, `option`, admin_menu_img, params)
```

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

[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 drop an email to feedback@packtpub.com, making sure to mention the book title in the subject of your message.

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

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

Customer Support

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

Downloading the Example Code for the Book

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

[The downloadable files contain instructions on how to use them.]

Errata

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

Questions

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

1 Joomla! Extension Development: An Overview

You have developed dynamic websites in the past, but a friend of yours told you about Joomla!, so you decide to give it a try. You wish to start a simple website about restaurants after being inspired by the attractive celebrity chefs from the Food Network. The installation goes smoothly and more quickly than attempting to build a content management system from scratch. After finding a delicious template, adding some menus, and banging out a couple of reviews, you begin to think of some of the features that will draw in more visitors and even some cash. Within minutes, you install a shopping cart for selling books, a forum for gathering suggestions of places to review, and some advertising affiliated programs for the sidebars.

However, as you glance through the homepage, you feel something is missing. Then suddenly a brilliant idea hits you for something entirely new. Once it is finished, you know others will want to use it for their sites as well. You look around Joomla!'s source files and start looking for a way of building code that will slide right into place.

Why Extend Joomla!

Joomla! is not only designed to handle the content articles, but also to allow a number of complex applications to be cleanly integrated. Shopping carts, forums, social networking profiles, job boards, and real estate listings are examples of extensions that the developers have written for Joomla!. All of these can run on a Joomla! site, and only a single database, template, and core need to be maintained. When you build an extension to Joomla!, it will inherit the look and feel of the overall site. Any type of program that can be coded in PHP is a potential component waiting to be written.

Your extensions can also be portable. When coded correctly, you will easily be able to install your code on another copy of Joomla! without having to enter the database logins and other basic configuration information again. Additionally, you will be able to distribute your extensions to others so that they can enjoy them, without any programming or database knowledge.

Customization versus Extension

Joomla!'s code is designed to be extended rather than hacked or directly modified. Rather than changing the core code, it is preferable to write an extension. When updates and patches are released for Joomla! itself, the core code will be updated, but your extensions will not be overwritten. These extensions are crafted in a self-contained manner, allowing you to freely develop your own code without disturbing other items present in the Joomla! installation.

Although they are self-contained, extensions do not operate in a completely sealed environment; you can mix different kinds to get the functionalities you desire. Joomla!'s code allows extensions to share resources and sometimes perform actions on each other. Since we can write extensions, we will do this instead of customizing the core.

How to Extend Joomla!

There are three types of extensions Joomla! supports, each with a specific use.

Components

Of the extensions available, components are the most essential. Components are essentially what you see in the "main" portion of the page. Joomla! is designed to load and run exactly one component for each page generated. Consequently, Joomla!'s core content management functionality is itself a component.

Components frequently have sophisticated back-end controls. The back end is commonly used to create and update records in database tables; also it can do typically anything, provided it is programmed in PHP. For instance, you may have a batch job that typically runs from a UNIX command line, but you can use the back end to provide a link where non-programmers can call it. You can also use it to allow site administrators to upload pictures or videos.

Modules

In contrast to components, any number of modules can appear on a page. Modules typically make up the elements of a sidebar or content menus. Modules complement the content contained in a component; they are not intended to be the main substance of a page. Joomla! also supports content modules, which involve no programming and can be displayed alongside coded components. The back-end controls for modules are limited, typically consisting of basic formatting.

Plug-Ins

When a piece of code is needed throughout the site, it is best implemented as a plug-in (formerly called a Mambot). Plug-ins are commonly used to format the output of a component or module when a page is built. Some examples of plug-ins include keyword highlighting, article comment boxes, and JavaScript-based HTML editors. Plug-ins can also be used to extend the results found in the core search component. The back-end controls are similar to those of modules.

Topic Overview

This book will cover the following topics regarding developing extensions for Joomla!.

Creating Toolbars and List Screens

Joomla! has a standard set of toolbar buttons used throughout the back end. These keep a consistent appearance across components, so users quickly become familiar with the corresponding functions. When necessary, the labeling and functions of these buttons can be changed and new buttons can also be added.

Like the standard toolbars, Joomla! has a certain look for screens that list a set of records from the database. These lists usually have links to edit screens for the individual records and have toggles that change the publishing status of the record. Automatic pagination is also available for lists.

Maintaining a Consistent Look and Reducing Repetitive Code Using HTML Functions

Several standard CSS class names are used to format content and HTML elements within your extensions. This makes it easy for your extensions to seamlessly blend in with the rest of the website. Additionally, Joomla! includes many functions to automate the generation of checkboxes, dropdowns, select lists, and other common elements.

Accessing the Database and Managing Records

A common database object is used in Joomla! so that only one connection is made during every page request. This object also provides a set of functions to make queries and retrieve results. These functions are database independent and are designed in such a way that you can install multiple copies of Joomla! into the same database when desired.

Besides a common database object, Joomla! has a standard database table class. Records can be created, read, updated, and deleted using the core functions. Logic can also be added so that child records in other tables are deleted when the parent is removed.

Security and the Preferred Way of Getting Request Variables

Since Joomla! is a web application deployed within public reach, it is necessary to protect it against security vulnerabilities. Joomla! employs a common method of making sure scripts are only called within the framework and not randomly executed.

Besides unintended script behavior, maliciously submitted data can be used by hackers to gain access to your database. Joomla! provides functionalities that prevent attacks of this kind.

Menu Item Control

A noteworthy feature of Joomla! is that navigation is separated from content. However, if a component is not built to take this into account, it is possible that website administrators will lose their template and module selections. To take advantage of the system, it is necessary to use the intended menu item ID number in generated links.

Also, it is possible to give administrators multiple options for linking to your component. This will allow the choice of different display options for the front end without the need to construct long, confusing URLs by hand. These options can additionally offer admins some simple configuration controls.

Controlling the Logic Flow Within a Component

The same file is always called when a certain component is loaded, but different functions are called within. Joomla! uses standard variables to determine which function to execute on each request. There are also classes available to automate the flow based on these variables.

At a minimum, components are designed to separate the output from the database and other processing functions. Larger components will separate the logic flow using a controller, the data access methods using a model, and the output using views. These conventions make it easier to maintain the code and help the component perform in a reliable and predictable way.

Configuration Through XML Parameters

Rather than creating a separate table to hold the configuration for an extension, Joomla! sets aside a place where short values can be held. These variables are defined through an XML file, which is installed with the extension. The XML file also provides default values and constraints for these parameters. Saving and retrieving of these values is automated; handwritten queries are not needed.

Packaging and Distributing

Once all of the code is complete, it is easily packaged for others to use. A listing of all the files involved is added to the XML file. Any queries needed for table creation are also included. All the files are then compressed in an archive. The extension is then ready to be installed on any Joomla!-based website.

Our Example Project

We will build extensions to create, find, promote, and cross-link restaurant reviews. A component will handle common data points seen across all reviews such as price range, reservations, cuisine type, and location. Your visitors will be able to search and sort the reviews, add their own criteria to zero in on their dining options for the evening.

Some modules will highlight new reviews, drawing the attention of frequent visitors. Finally, one plug-in will pull pieces of the reviews into feature articles and another will integrate them into searches.

To prepare for this project, install a fresh copy of Joomla! 1.5 on a web server with PHP and a database (preferably MySQL). If you prefer to exclusively use one computer to complete this project and do not have a local web server, it will probably be easier to download and install a bundled and pre-configured package such as XAMPP (`http://www.apachefriends.org`). In this way you will be able to work with all the files on your local file system.

Summary

Joomla! can be extended through components, modules, and plug-ins. This allows you to add functionalities to a Joomla! site without hacking the core code. Joomla! can then be maintained and updated without disturbing the custom code.

2
Getting Started with Component Development

Before you begin with coding, there are a few files and folders that have to be created, and also a query that has to be run. This will not only allow you to build the components but also help you try different features without extensive configuration. You will also get a quick overview of the way components are organized and accessed through Joomla!. Finally, you will add some toolbars that work just like those in other components.

Joomla!'s Component Structure

Joomla! employs a specific naming scheme, which is used by all components. Each component in the system has a unique name with no spaces. The code is split into two folders, each bearing the component name prefixed by `com_`. The component in this book will be called `reviews`. Therefore, you will have to create two folders named `com_reviews`:

- Create one in the folder named `components` for the front end.
- Create one in the folder named `components` within the `administrator` folder for the back end.

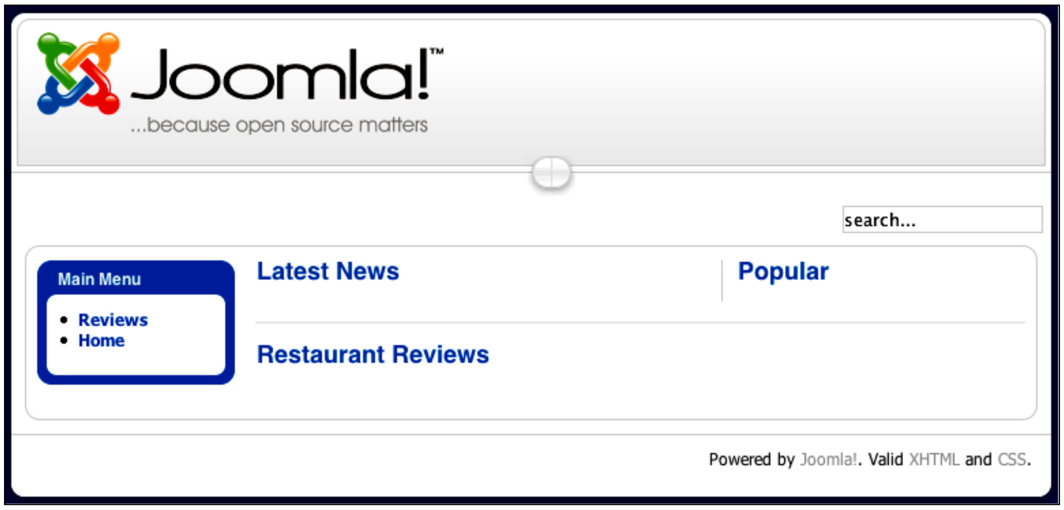
When the component is loaded from the front end, Joomla! will look for a file with the component's unique name ending in a `.php` extension. Within the `components/com_reviews` folder, create the `reviews.php` file. Similarly, running it in the back end assumes the presence of a file prefaced with `admin.` followed by the component name and ending in `.php`. Add the file `admin.reviews.php` in `administrator/components/com_reviews`. Leave both the files empty for the moment.

Executing the Component

All front-end requests in Joomla! go through `index.php` in the root directory. Different components can be loaded by setting the `option` variable in the URL GET string. If you install Joomla! on a local web server in a directory titled `joomla`, the URL for accessing the site will be `http://localhost/joomla/index.php` or something similar. Assuming this is the case, you can load the component's front end by opening `http://localhost/joomla/index.php?option=com_reviews` in your browser. At this point, the screen should be essentially blank, apart from the common template elements and modules. To make this component slightly more useful, open `reviews.php` and add the following code, then refresh the browser:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
echo '<div class="componentheading">Restaurant Reviews</div>';
?>
```

Your screen will look similar to the following:

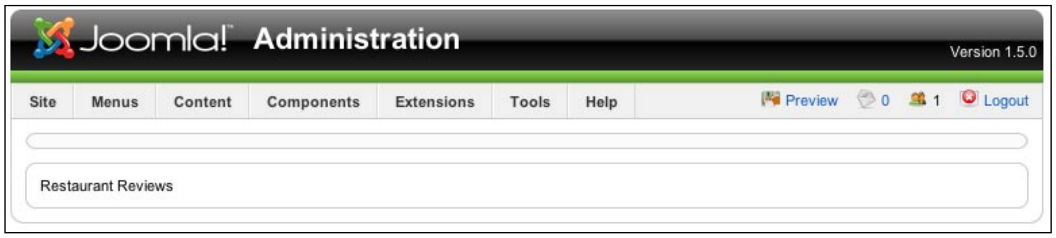


You may be wondering why we called `defined()` at the beginning of the file. This is a check to ensure that the code is called through Joomla! instead of being accessed directly at `components/com_reviews/reviews.php`. Joomla! automatically configures the environment with some security safeguards that can be defeated if someone is able to directly execute the code for your component.

For the back end, drop this code into `administrator/components/com_reviews/admin.reviews.php`:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
echo 'Restaurant Reviews';
?>
```

Go to `http://localhost/joomla/administrator/index.php?option=com_reviews` and compare your result to this:

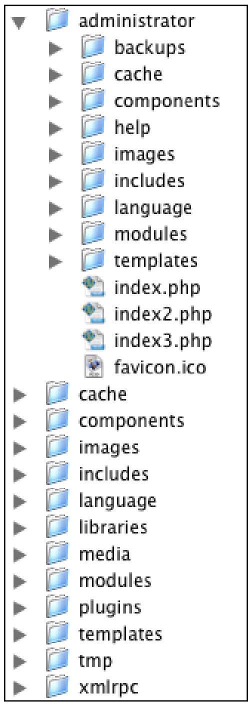


Restaurant Reviews

Joomla!'s Division between Front End and Back End

For all Joomla! components, code empowering the back-end portion is kept away from the front-end code. In some instances, such as the database table class, the back end will use certain files from the front end, but otherwise the two are separate. Security is enhanced as you are less likely to slip the administrative functions into the front-end code. This is an important distinction as the front end and back end are similar in structure.

The following folder diagram shows the Joomla! root with the `administrator` folder expanded:



Notice that the `administrator` folder has a structure similar to the root folder. It is important to differentiate between the two, else you may place your code in the wrong folder and it will fail to execute until you move it.

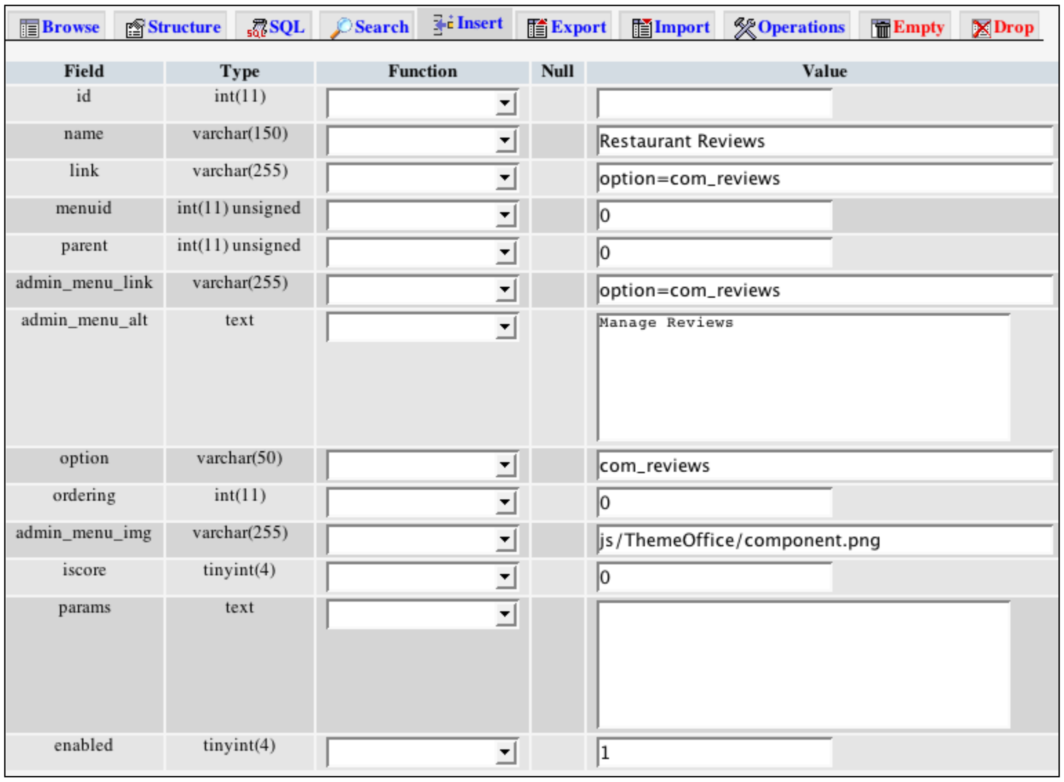
Registering Your Component in the Database

You now know how to access both the front end and back end of the component. Although you could keep typing in the URLs each time you wanted to execute a piece of code, this will not be acceptable to your users. Navigation can be provided if you register the component in the database by adding a row to the components table.

We will perform this registration using the following query. It is assumed that your database prefix is `jos_`. If not, replace `jos_` with the prefix you chose. If you prefer to work with direct SQL statements on a command-line interface, enter the following query in your console:

```
INSERT INTO jos_components (name, link, admin_menu_link,
            admin_menu_alt, 'option', admin_menu_img, params)
VALUES ('Restaurant Reviews', 'option=com_reviews',
        'option=com_reviews', 'Manage Reviews', 'com_reviews',
        'js/ThemeOffice/component.png', '');
```

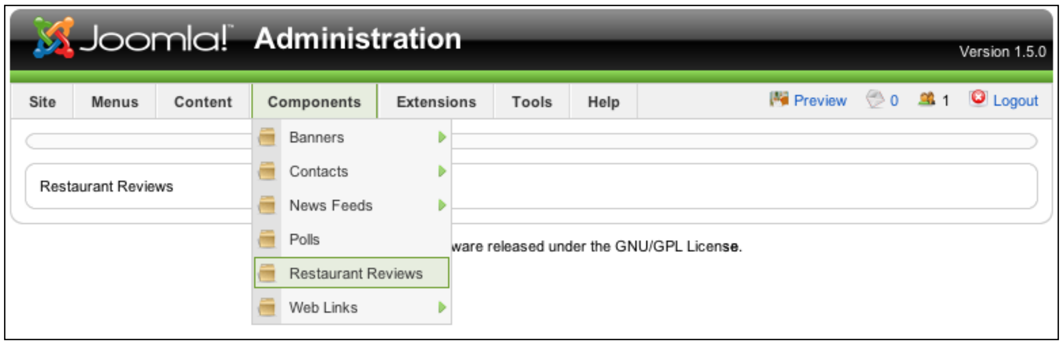
If you prefer to use a GUI or web-based database manager such as phpMyAdmin, enter **Restaurant Reviews** for **name**, **option=com_reviews** for **link** and **admin_menu_link**, **Manage Reviews** for **admin_menu_alt**, **com_reviews** for **option**, and **js/ThemeOffice/component.png** for **admin_menu_img**. Leave all of the other fields blank. The fields **menuid**, **parent**, **ordering**, and **iscore** will default to **0**, while **enabled** will default to **1**.



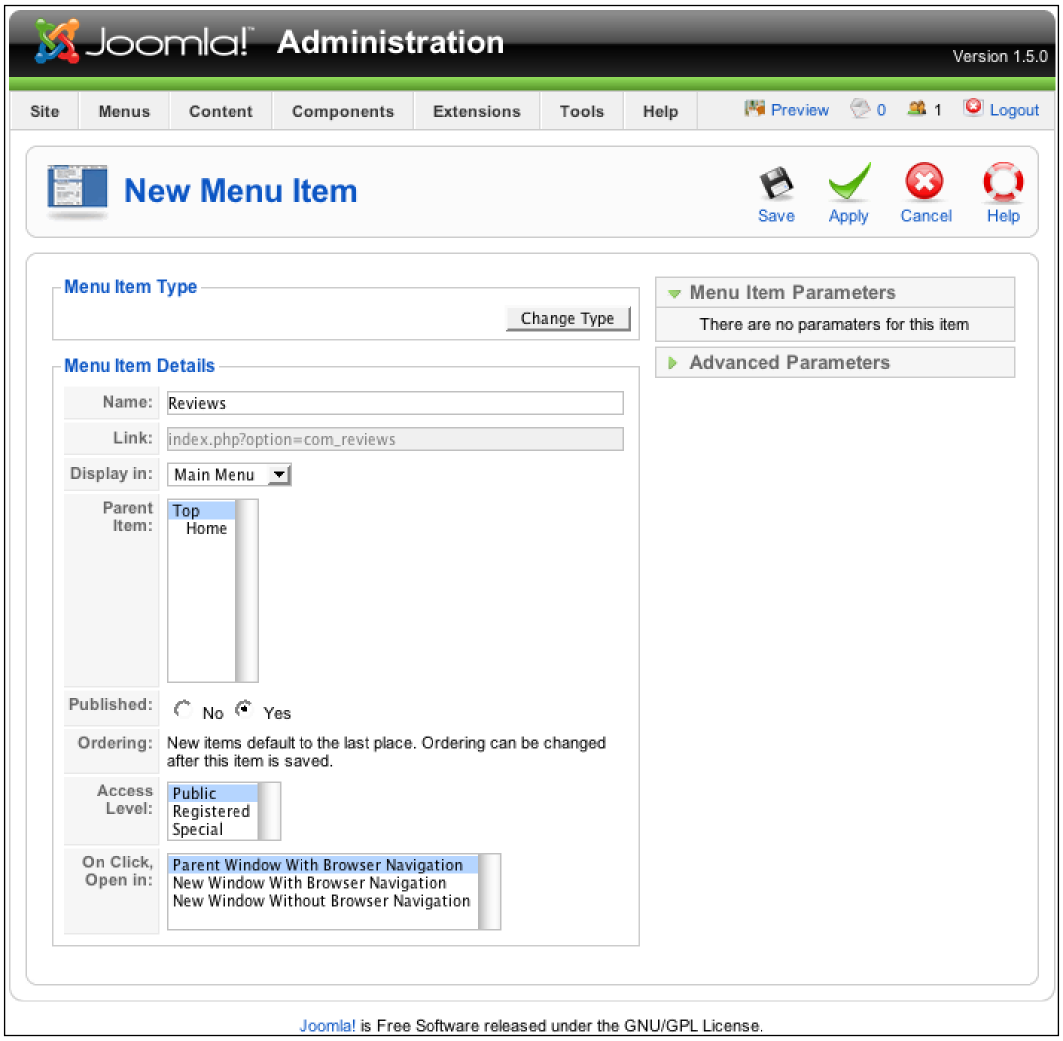
Field	Type	Function	Null	Value
id	int(11)			
name	varchar(150)			Restaurant Reviews
link	varchar(255)			option=com_reviews
menuid	int(11) unsigned			0
parent	int(11) unsigned			0
admin_menu_link	varchar(255)			option=com_reviews
admin_menu_alt	text			Manage Reviews
option	varchar(50)			com_reviews
ordering	int(11)			0
admin_menu_img	varchar(255)			js/ThemeOffice/component.png
iscore	tinyint(4)			0
params	text			
enabled	tinyint(4)			1

Adding this record gives the system some basic information about your component. It states the name you want to use for the component, which can contain spaces and punctuation. You can put in specific links to go to both the front end and back end. The image to be used on the **Components** menu can be specified. Also as the description in the browser status bar can be made available. It is not necessary to add this query while developing the component; once you create the basic directories and files, your component is ready to be executed. However, it does add a menu item in the back end and makes it possible to add an appropriate link in the front end without hard coding a URL.

After the record is successfully entered, go to any page in the back end and refresh it. When you move the mouse cursor over the **Components** menu you should see the new option:



Now that the component is registered, you can also create a link for the front end. Go to **Menus | Main Menu** and click **New**. From this screen, select **Restaurant Reviews**. Enter **Reviews** as the **Name**. The following screen will be observed:

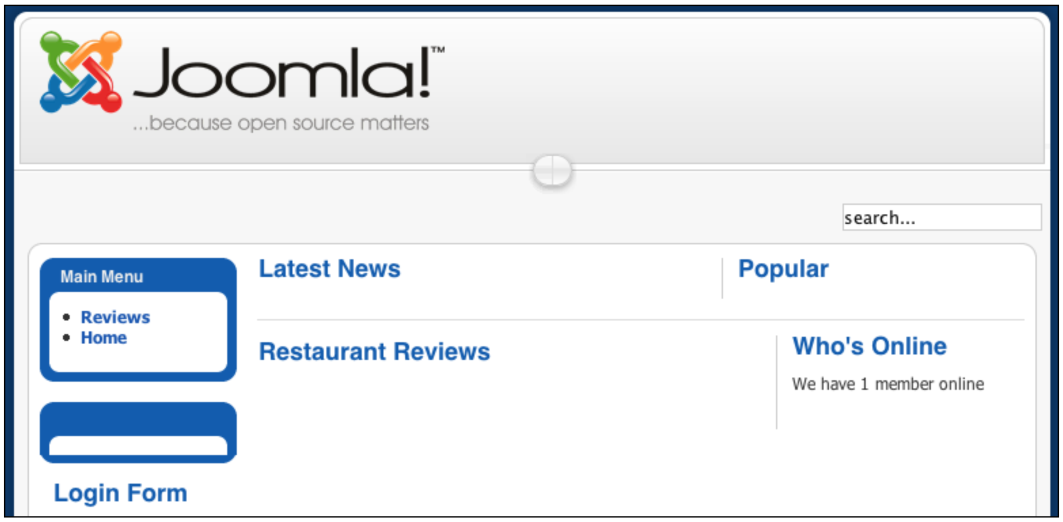


There are no paramaters for this item

Name:	Reviews
Link:	index.php?option=com_reviews
Display in:	Main Menu
Parent Item:	Top Home
Published:	No [✓ Yes]
Ordering:	New items default to the last place. Ordering can be changed after this item is saved.
Access Level:	Public Registered Special
On Click, Open in:	Parent Window With Browser Navigation New Window With Browser Navigation New Window Without Browser Navigation

Joomla! is Free Software released under the GNU/GPL License.

Now click **Save** and go to the front end. You should now see **Reviews** listed as an option.



You could just break out your PHP skills and start coding the component, ensuring all front-end requests go through `http://localhost/joomla/index.php?option=com_reviews` and all back-end requests go though `http://localhost/joomla/administrator/index.php?option=com_reviews`. Joomla! is flexible enough to let you do as you please. In some cases, you will have existing code that you may want to use and you will need to split it into appropriate files. But for the restaurant reviews, you will start a new Joomla! component from scratch. You have the opportunity to design everything with Joomla's toolbars, users, database classes, and libraries in mind. These elements will save you a lot of time once you understand how they work.

Creating Toolbars

Throughout the back end, all the core components implement toolbars with similar buttons for saving, deleting, editing, and publishing items. You can use these buttons in your component so that frequent administrators will have a seamless experience.

To start, create the `toolbar.reviews.html.php` file in the `administrator/components/com_reviews folder` and enter in the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
class TOOLBAR_reviews {
  function _NEW() {
    JToolBarHelper::save();
    JToolBarHelper::apply();
    JToolBarHelper::cancel();
  }

  function _DEFAULT() {
    JToolBarHelper::title( JText::_( 'Restaurant Reviews' ),
                                           'generic.png' );
    JToolBarHelper::publishList();
    JToolBarHelper::unpublishList();
    JToolBarHelper::editList();
    JToolBarHelper::deleteList();
    JToolBarHelper::addNew();
  }
}
?>
```

Files containing output codes are usually organized into classes, like the code here with `TOOLBAR_reviews`. Each member function here represents a different toolbar. The class `JToolBarHelper` contains functions that generate all the HTML necessary to build toolbars. When desired, you can also add custom HTML output from within these functions. Be aware that the toolbars lie within HTML tables; you will probably want to add `<td>` tags along with your custom navigation.

The toolbars are now defined, but you need to add some code that will decide which one to display. In the back end, Joomla! automatically loads the file beginning with the component name and ending in `.reviews.php` in the upper right-hand portion of the screen. Add the following code into `toolbar.reviews.php` in the `administrator/components/com_reviews` folder:

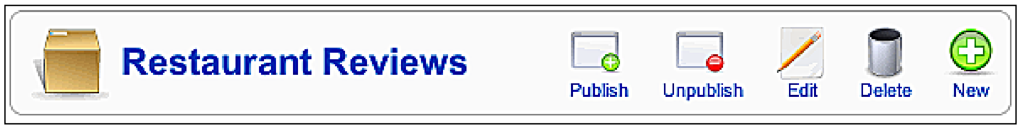
```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
require_once( JApplicationHelper::getPath( 'toolbar_html' ) );
switch($task)
{
  case 'edit':
  case 'add':
    TOOLBAR_reviews::_NEW();
    break;

  default:
    TOOLBAR_reviews::_DEFAULT();
    break;
}
?>
```

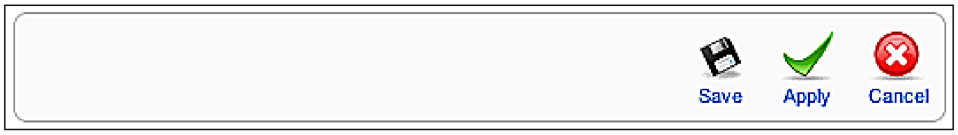
The line containing `require_once(...)` uses the `getPath()` member function of the `JApplicationHelper` class. The call to `getPath()` allows you to call up the `toolbar.reviews.html.php` file without committing to a component name. Later, even if you change the name to 'Restaurants' and also change the filenames, this line of code will still load the output code for the toolbar with no modification.

You may be wondering why we are creating two files to begin with, `toolbar.reviews.php` and `toolbar.reviews.html.php`. The preferred coding style among component developers is to keep the processing logic in a file completely separate from where the actual output takes place. This makes it easier to add features later and to potentially share the code with others.

After `toolbar.reviews.php` loads the file with the output class, you need to decide which toolbar should be displayed. The request variable `$task` is automatically registered in global scope by Joomla! and is used to direct the logic flow within the component. With your toolbar code in place, refresh the browser in the back end and go to **Restaurant Reviews** under **Components** and you should see the following screen:



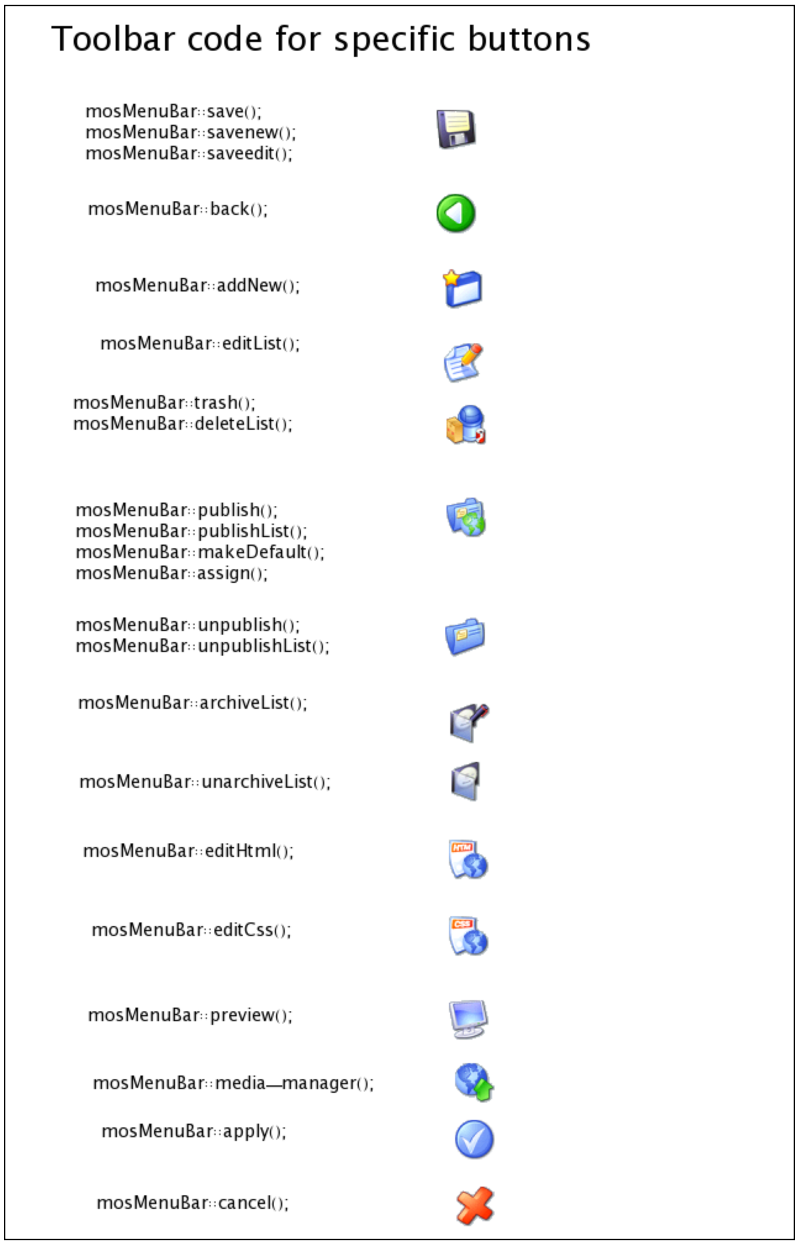
To see the other toolbar, add `&task=add` to the end of the URL in your browser, then load it. The toolbar should appear like this:



Your users will certainly not want to add the task variable at the end of the URL as they navigate through your component. How will they be able to use the second toolbar then? Each button on the toolbar represents a different task. When one is clicked, the associated task is added to your form and it is automatically submitted. Once the appropriate form is in place, a click on the **New** button from the first screen will pull up the toolbar seen in the second. Since we do not yet have any forms in the back end, these toolbar buttons will not function. These will start working in the next chapter when we build out the rest of the back end.

Available Toolbar Buttons

Joomla! allows you to override any button with your own task and label, passing them as the first and second parameters respectively. The following buttons are available with the standard distribution of Joomla!:



Toolbar code for specific buttons

```
mosMenuBar::save();
mosMenuBar::savenew();
mosMenuBar::saveedit();

mosMenuBar::back();

mosMenuBar::addNew();

mosMenuBar::editList();

mosMenuBar::trash();
mosMenuBar::deleteList();

mosMenuBar::publish();
mosMenuBar::publishList();
mosMenuBar::makeDefault();
mosMenuBar::assign();

mosMenuBar::unpublish();
mosMenuBar::unpublishList();

mosMenuBar::archiveList();

mosMenuBar::unarchiveList();

mosMenuBar::editHtml();

mosMenuBar::editCss();

mosMenuBar::preview();

mosMenuBar::media_manager();

mosMenuBar::apply();

mosMenuBar::cancel();
```

If you would like to create a custom button that looks and behaves like the core ones, use the `custom()` member function of `JToolBarHelper`, passing in the task, icon, mouse-over image, and text description as the respective parameters.

Summary

The basic files necessary to build the component are now in place. The rest of the Joomla! installation now knows that this component is available for front end and back end use. By using standard HTML and CSS classes, the component has a look and feel similar to the other components in the system, making it easy to use with different templates. Basic toolbars are available to the component and can be assigned to different screens by using the `$task` variable.

3
Back-End Development

Creating and managing reviews is our component's largest task. We will add forms and database functions to take care of this so that we can start adding reviews. This will also give us a chance to allow some of our restaurant reviewers to offer feedback. We will cover the following topics in this chapter:

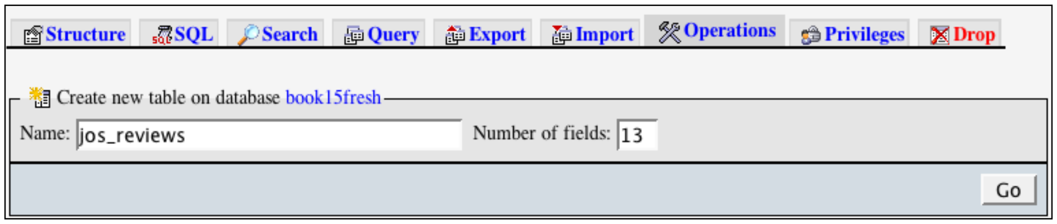
- Creating a database table to hold the reviews
- Setting up a basic form for data entry
- Processing the data and adding it to the database
- Listing the existing reviews
- Editing and deleting reviews

Creating the Database Table

Before we set up an interface for entering reviews, we need to create a place in the database where they will go. We will start with a table where one row will represent one review in the system. Assuming that your database prefix is `jos_` (check **Site | Configuration | Server** if you are unsure), enter the following query into your SQL console:

```
CREATE TABLE 'jos_reviews'
(
  'id' int(11) NOT NULL auto_increment,
  'name' varchar(255) NOT NULL,
  'address' varchar(255) NOT NULL,
  'reservations' varchar(31) NOT NULL,
  'quicktake' text NOT NULL,
  'review' text NOT NULL,
  'notes' text NOT NULL,
  'smoking' tinyint(1) unsigned NOT NULL default '0',
  'credit_cards' varchar(255) NOT NULL,
  'cuisine' varchar(31) NOT NULL,
  'avg_dinner_price' tinyint(3) unsigned NOT NULL default '0',
  'review_date' datetime NOT NULL,
  'published' tinyint(1) unsigned NOT NULL default '0',
  PRIMARY KEY  ('id')
);
```

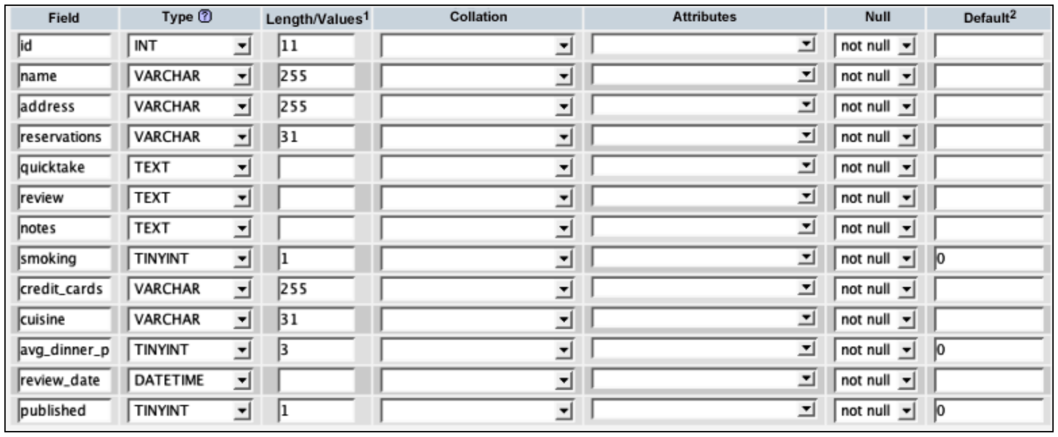
If you're using phpMyAdmin, pull up the following screen and enter **jos_reviews** as the table name and let it generate **13** fields:



Create new table on database book15fresh

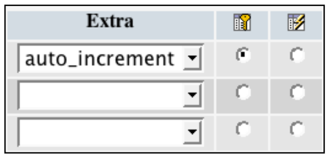
Name: jos_reviews   Number of fields: 13

After clicking **Go**, you will see a grid; fill in details so that it looks like the following screen:



Field	Type	Length/Values[1]	Collation	Attributes	Null	Default[2]
id	INT	11			not null	
name	VARCHAR	255			not null	
address	VARCHAR	255			not null	
reservations	VARCHAR	31			not null	
quicktake	TEXT				not null	
review	TEXT				not null	
notes	TEXT				not null	
smoking	TINYINT	1			not null	0
credit_cards	VARCHAR	255			not null	
cuisine	VARCHAR	31			not null	
avg_dinner_p	TINYINT	3			not null	0
review_date	DATETIME				not null	
published	TINYINT	1			not null	0

Be sure you make the field **id** into an automatically incremented primary key:



Creating a Table Class

We could write individual functions to take care of the queries necessary to add, update, and delete the reviews. However, these are rudimentary functions that you would prefer not to write. Fortunately, the Joomla! team has already done this for you. The `JTable` class provides functions for creating, reading, updating, and deleting records from a single table in the database.

To take advantage of `JTable`, we need to write an extension of it specific to `jos_reviews`. In the `/administrator/components/com_reviews` folder, create a folder named `tables`. In this folder, create the `review.php` file and enter the following code:

```
<?php
defined('_JEXEC') or die('Restricted access');
class TableReview extends JTable
{
  var $id = null;
  var $name = null;
  var $address = null;
  var $reservations = null;
  var $quicktake = null;
  var $review = null;
  var $notes = null;
  var $smoking = null;
  var $credit_cards = null;
  var $cuisine = null;
  var $avg_dinner_price = null;
  var $review_date = null;
  var $published = null;
  function __construct(&$db)
  {
    parent::__construct( '#__reviews', 'id', $db );
  }
}
?>
```

When we extend the `JTable` class, we add all the columns of the database table as member variables and set them to **null.** Also we override the class constructor: the `__construct()` method. At the minimum, our `__construct()` method will take a database object as a parameter and will call the parent constructor using the name of the database table (where `#__` is the table prefix), the primary key, and the database object.

Why use #__ as the Table Prefix?

When writing queries and defining `JTable` extensions in Joomla!, use `#__` instead of `jos_`. When Joomla! executes the query, it automatically translates `#__` into the database prefix chosen by the admin. This way, someone can safely run multiple installations of Joomla! from the same database. This also makes it possible for you to change the prefix to anything you like without changing the code. You can hard-code the names of legacy tables that cannot be renamed to follow this convention, but you will not be able to offer the multiple installation compatibility.

The `TableReview` class inherits the `bind()`, `store()`, `load()`, and `delete()`, functions among others. These four functions allow you to manage records in the database without writing a single line of SQL.

Creating the Review Form

With a database table now in place, we need a friendly interface for adding reviews into it. To start, let's create a form for entering the review data. As we did with the toolbar files, we want to separate the HTML output from our processing logic. The PHP code necessary for configuring the form will be in `admin.reviews.php` while `admin.reviews.html.php` will contain the actual HTML output. Open `admin.reviews.php` and replace the contents with the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
require_once( JApplicationHelper::getPath( 'admin_html' ) );
JTable::addIncludePath(JPATH_COMPONENT.DS.'tables');
switch($task)
{
  case 'add':
  editReview( $option );
    break;
}
function editReview( $option )
{
  $row =& JTable::getInstance('Review', 'Table');
  $lists = array();
  $reservations = array(
    '0' => array('value' => 'None Taken',
           'text' => 'None Taken'),
    '1' => array('value' => 'Accepted',
           'text' => 'Accepted'),
    '2' => array('value' => 'Suggested',
           'text' => 'Suggested'),
    '3' => array('value' => 'Required',
           'text' => 'Required'),
  );
  $lists['reservations'] = JHTML::_('select.genericList',
  $reservations, 'reservations', 'class="inputbox" '. '', 'value',
                                       'text', $row->reservations );
  $lists['smoking'] = JHTML::_('select.booleanlist', 'smoking',
                              'class="inputbox"', $row->smoking);
  $lists['published'] = JHTML::_('select.booleanlist', 'published',
                            'class="inputbox"', $row->published);
  HTML_reviews::editReview($row, $lists, $option);
}
?>
```

After checking to make sure we're within Joomla!, we use `require_once( JApplicationHelper::getPath( 'admin_html' ) )` to include `admin.reviews.html.php`. The `getPath()` function takes certain strings (such as `admin_html`, `front_html`, and `class`) and returns the absolute path to the corresponding component files. Although we haven't specified the component name in this line of code, it will still include the appropriate file, even if we change the name of the component and the HTML file to something else. Using `require_once()` ensures the file is added only once.

Although we won't be working with the database right away, we do want to include our `table` class. This is accomplished through the `addIncludePath()` member function of `JTable`. The `addIncludePath()` function automatically includes all the classes we've defined in files in the `tables` directory. The filename and path are constructed to be cross-platform compatible. Joomla! sets `JPATH_COMPONENT` to the absolute path of the back-end code. The constant `DS` is the operating-system-specific directory separator to be used.

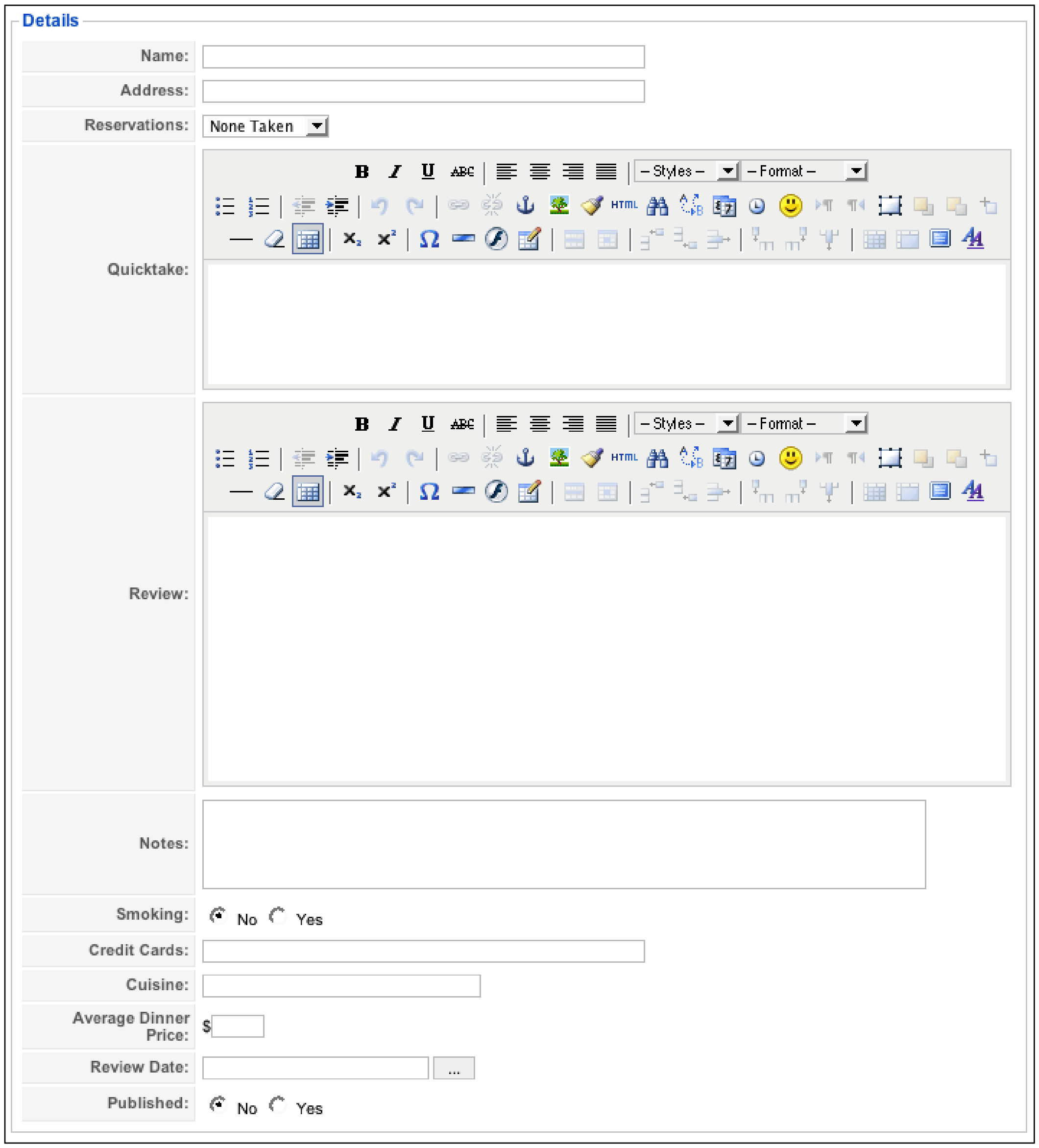
The `switch()` statement checks the `$task` variable and chooses an appropriate function to run based on the value. Finally, the `editReview()` function prepares a few HTML elements before passing them along to our display function `HTML_reviews::editReview()`.

Now create the `admin.reviews.html.php` file and add the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
class HTML_reviews
{
  function editReview( $row, $lists, $option )
  {
    $editor =& JFactory::getEditor();
    JHTML::_('behavior.calendar');
    ?>
    <form action="index.php" method="post"
                 name="adminForm" id="adminForm">
      <fieldset class="adminform">
        <legend>Details</legend>
        <table class="admintable">
        <tr>
          <td width="100" align="right" class="key">
            Name:
          </td>
          <td>
            <input class="text_area" type="text" name="name"
               id="name" size="50" maxlength="250"
               value="<?php echo $row->name;?>" />
          </td>
        </tr>
        <tr>
          <td width="100" align="right" class="key">
            Address:
          </td>
          <td>
            <input class="text_area" type="text" name="address"
               id="address" size="50" maxlength="250"
               value="<?php echo $row->address;?>" />
          </td>
        </tr>
        <tr>
          <td width="100" align="right" class="key">
            Reservations:
          </td>
          <td>
            <?php
            echo $lists['reservations'];
            ?>
          </td>
        </tr>
        <tr>
          <td width="100" align="right" class="key">
            Quicktake:
          </td>
          <td>
            <?php
            echo $editor->display( 'quicktake',  $row->quicktake ,
                                      '100%', '150', '40', '5' ) ;
            ?>
          </td>
        </tr>
        <tr>
          <td width="100" align="right" class="key">
            Review:
          </td>
          <td>
            <?php
            echo $editor->display( 'review',  $row->review ,
                                     '100%', '250', '40', '10' ) ;
            ?>
          </td>
        </tr>
        <tr>
          <td width="100" align="right" class="key">
            Notes:
          </td>
          <td>
            <textarea class="text_area" cols="20" rows="4"
               name="notes" id="notes" style="width:500px"><?php echo
               $row->notes; ?></textarea>
          </td>
        </tr>
        <tr>
          <td width="100" align="right" class="key">
            Smoking:
          </td>
          <td>
            <?php
            echo $lists['smoking'];
            ?>
```

```
            </td>
          </tr>
          <tr>
            <td width="100" align="right" class="key">
              Credit Cards:
            </td>
            <td>
              <input class="text_area" type="text" name="credit_cards"
                 id="credit_cards" size="50" maxlength="250"
                 value="<?php echo $row->credit_cards;?>" />
            </td>
          </tr>
          <tr>
            <td width="100" align="right" class="key">
              Cuisine:
            </td>
            <td>
              <input class="text_area" type="text" name="cuisine"
                 id="cuisine" size="31" maxlength="31"
                 value="<?php echo $row->cuisine;?>" />
            </td>
          </tr>
          <tr>
            <td width="100" align="right" class="key">
              Average Dinner Price:
            </td>
            <td>
              $<input class="text_area" type="text"
                   name="avg_dinner_price"
                   id="avg_dinner_price" size="5" maxlength="3"
                   value="<?php echo $row->avg_dinner_price;?>" />
            </td>
          </tr>
          <tr>
            <td width="100" align="right" class="key">
              Review Date:
            </td>
            <td>
              <input class="inputbox" type="text" name="review_date"
                 id="review_date" size="25" maxlength="19"
                 value="<?php echo $row->review_date; ?>" />
              <input type="reset" class="button" value="..."
                 onclick="return showCalendar('review_date',
                 'y-mm-dd');" />
            </td>
          </tr>
          <tr>
            <td width="100" align="right" class="key">
              Published:
            </td>
            <td>
              <?php
              echo $lists['published'];
              ?>
            </td>
          </tr>
          </table>
        </fieldset>
        <input type="hidden" name="id"
          value="<?php echo $row->id; ?>" />
        <input type="hidden" name="option"
          value="<?php echo $option;?>" />
        <input type="hidden" name="task"
          value="" />
      </form>
      <?php
    }
  }
  ?>
```

Point your browser to `http://localhost/joomla/administrator/index.php?option=com_reviews&task=add` and you should see the following screen:



Details

Name:	
Address:	
Reservations:	None Taken
Quicktake:	
Review:	
Notes:	
Smoking:	[✓ No] Yes
Credit Cards:	
Cuisine:	
Average Dinner Price:	$
Review Date:	...
Published:	[✓ No] Yes

Our `editReview()` function takes in a database table row object, an array of HTML snippets, and the name of the component to generate this screen. This way, `editReview()` is almost entirely devoted to the output. Before giving the output, the function does include a couple of pieces of helper code to power some of the UI elements.

What does JTML::_() do?

Joomla! includes many HTML generation functions that can be used to automate the creation of elements such as dropdown lists and checkboxes. In an effort to speed performance, these functions are only loaded into memory when needed. This is accomplished though the `_()` function, which takes the function name as the first parameter and passes the remaining parameters (if any) to the desired function. Functions are organized by type, which is indicated by the first part of the name passed into the first parameter of `_()` before the period.

First we pull in an object representing the admin's HTML editor of choice with the `JFactory::getEditor();` function. Just beneath this, we also use `JHTML::_('behavior.calendar')` to add JavaScript and CSS includes to the header; these are necessary for the pop-up calendar on the **Review Date** field:

```
class HTML_reviews
{
  function editReview( $row, $lists, $option )
  {
    $editor =& JFactory::getEditor();
    JHTML::_('behavior.calendar');
```

The `display()` member function of the editor object returns HTML for the chosen rich text editor. If no rich text editor is desired, this will return a `<textarea>` field instead.

```
            <td>
            <?php
            echo $editor->display( 'quicktake',  $row->quicktake ,
                '100%', '150', '40', '5' ) ;
            ?>
          </td>
```

The `display()` member function takes the following elements: form variable name, value, width, height, columns, and rows. The last two are for `<textarea>` dimensions when the admin has opted not to use an HTML editor.

Processing the Data

Once the data in the form is filled out and the admin clicks the **Save** button, we need to save the information into the database. To start, create `saveReview()` in `admin.reviews.php`:

```
function saveReview( $option )
{
  global $mainframe;
  $row =& JTable::getInstance('review', 'Table');
  if (!$row->bind(JRequest::get('post')))
  {
    echo "<script> alert('".$row->getError()."');
    window.history.go(-1); </script>\n";
    exit();
  }
  $row->quicktake = JRequest::getVar( 'quicktake', '', 'post',
                                      'string', JREQUEST_ALLOWRAW );
  $row->review = JRequest::getVar( 'review', '', 'post',
                                      'string', JREQUEST_ALLOWRAW );
  if(!$row->review_date)
    $row->review_date = date( 'Y-m-d H:i:s' );
  if (!$row->store())
  {
    echo "<script> alert('".$row->getError()."');
    window.history.go(-1); </script>\n";
    exit();
  }
  $mainframe->redirect('index.php?option=' .
                       $option, 'Review Saved');
}
```

First, we pull in the global `$mainframe` and the current database connection. The `$mainframe` object has many member functions you can use to control session variables and headers. We also set `$row` as an instance of our `TableReview` class; the name of the class is assembled from the parameters, with the second acting as a prefix for the first. Next, we call the `bind()` member function of `$row` to load in all of the variables from the form.

The `bind()` function takes an associative array as the parameter and attempts to match all of the elements to member variables of the object. To reduce the risk of SQL injection attacks, we call `JRequest::get()` to sanitize the values from `$_POST`. This process escapes characters that could be used to control the SQL query.

If `bind()` fails for some reason, we display this as a JavaScript alert and take the user back to the previous screen.

After binding, we can manipulate the member variables of `$row` directly. Since the `quicktake` and `review` fields accept HTML content, they need special handling as the `bind()` function automatically strips out HTML. To get around this, we use the `getVar()` member function of `JRequest`, passing in the form variable name, the default value, the request array we wish to pull from, the expected format, and the `JREQUEST_ALLOWRAW` flag respectively.

In addition to recaputring the HTML data, we are also able to add default data or some other automatically generated data after binding. We've set it to fill in the current date for the review in case it was not chosen.

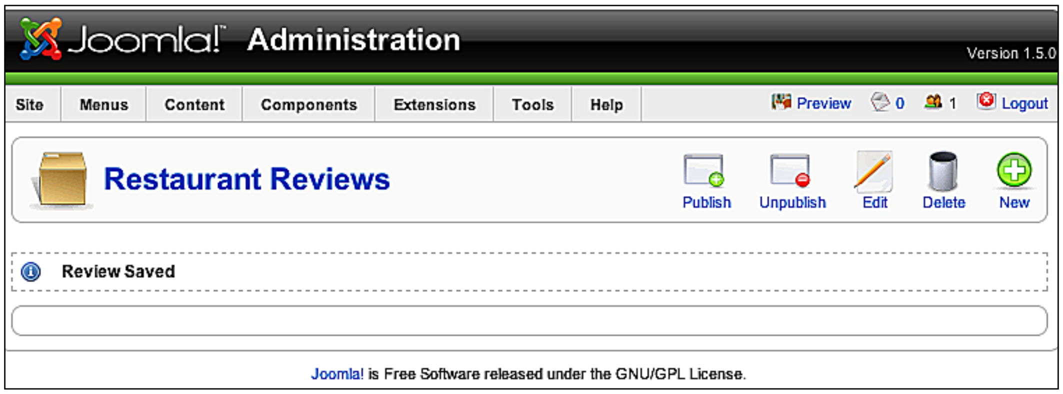
Finally, we call the `store()` function, which takes all the member variables and turns them into an `UPDATE` or `INSERT` statement, depending on the value of **id**. Since we are creating this record for the first time, it will not have a value for **id** and so an `INSERT` query will be constructed.

If there is an SQL error, we return it back to the user and return the user back to the previous screen. Frequently, SQL errors at this level can be caused by extraneous member variables of `$row` not present in the `table` class. If you run into a query error, first check to make sure the spelling of your member variables matches the spelling of the table columns. Otherwise, if the SQL is successful, we use the `redirect()` function from `$mainframe` to send the user back to the main component screen with a confirmation message.

At the moment, the `switch()` statement in `admin.reviews.php` only processes the `add` task. Now that we have a form and function in place, we can add a case to save our data. Add the highlighted code below to the `switch()`:

```
switch($task)
{
  case 'add':
  editReview( $option );
    break;
  case 'save':
  saveReview( $option );
    break;
}
```

Save all your files and go to `http://localhost/joomla/administrator/index.php?option=com_reviews&task=add` in your browser. You should now be able to fill out the form and click **Save**. You should see a screen similar to the following:



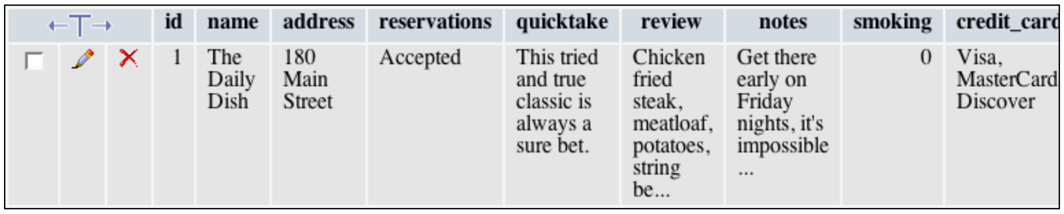
Review Saved

Joomla! is Free Software released under the GNU/GPL License.

Why Can't I Click the 'New' Button?

The buttons on the toolbar are designed to work with the form named `adminForm`. Since this screen does not have a form yet, clicking on any of the buttons will result in a JavaScript error. Once `adminForm` is added with the hidden variable `task`, the buttons will function as expected.

You can check the results in the `jos_reviews` database table. If everything works correctly, a table listing in phpMyAdmin shows the result after you click on **Browse**.



←T→			id	name	address	reservations	quicktake	review	notes	smoking	credit_card
			1	The Daily Dish	180 Main Street	Accepted	This tried and true classic is always a sure bet.	Chicken fried steak, meatloaf, potatoes, string be...	Get there early on Friday nights, it's impossible ...	0	Visa, MasterCard Discover

Creating a List Screen

Since our admins will not have access to phpMyAdmin, we need to build a screen that lists all of the reviews in the database. To start, add the following function to `admin.reviews.php`:

```
function showReviews( $option )
{
  $db =& JFactory::getDBO();
  $query = "SELECT * FROM #__reviews";
  $db->setQuery( $query );
  $rows = $db->loadObjectList();
  if ($db->getErrorNum()) {
      echo $db->stderr();
      return false;
  }
  HTML_reviews::showReviews( $option, $rows );
}
```

This function loads the data to be displayed, so we get a reference to the current database connection, then call its member function `setQuery()`. The `setQuery()` function takes the string of a query and stores it for later use, rather than executing it right away. When we call `loadObjectList()`, the previously set query is run and the individual rows in the result are loaded into an array as objects. If we run into an error, we display it and stop the component.

If all goes well, we pass the array of results into the following member function to be added to `admin.reviews.html.php`:

```
function showReviews( $option, &$rows )
{
  ?>
  <form action="index.php" method="post" name="adminForm">
  <table class="adminlist">
    <thead>
      <tr>
        <th width="20">
          <input type="checkbox" name="toggle"
               value="" onclick="checkAll(<?php echo
               count( $rows ); ?>);" />
        </th>
        <th class="title">Name</th>
        <th width="15%">Address</th>
        <th width="10%">Reservations</th>
        <th width="10%">Cuisine</th>
        <th width="10%">Credit Cards</th>
        <th width="5%" nowrap="nowrap">Published</th>
      </tr>
    </thead>
```

```
      <?php
      $k = 0;
      for ($i=0, $n=count( $rows ); $i < $n; $i++)
      {
        $row = &$rows[$i];
        $checked = JHTML::_('grid.id', $i, $row->id );
        $published = JHTML::_('grid.published', $row, $i );
        ?>
        <tr class="<?php echo "row$k"; ?>">
          <td>
            <?php echo $checked; ?>
          </td>
          <td>
            <?php echo $row->name; ?>
          </td>
          <td>
            <?php echo $row->address; ?>
          </td>
          <td>
            <?php echo $row->reservations; ?>
          </td>
          <td>
            <?php echo $row->cuisine; ?>
          </td>
          <td>
            <?php echo $row->credit_cards; ?>
          </td>
          <td align="center">
            <?php echo $published;?>
          </td>
        </tr>
        <?php
        $k = 1 - $k;
      }
      ?>
    </table>
    <input type="hidden" name="option"
                      value="<?php echo $option;?>" />
    <input type="hidden" name="task" value="" />
    <input type="hidden" name="boxchecked" value="0" />
    </form>
    <?php
  }
```

This function starts by defining a form that points to `index.php`, with the name set to **adminForm (**for JavaScript references). A table with the `adminlist` class is then started and headers are added. All the headers are typical, except for the first one that acts as a "check all" checkbox that automatically selects all the records on the screen.

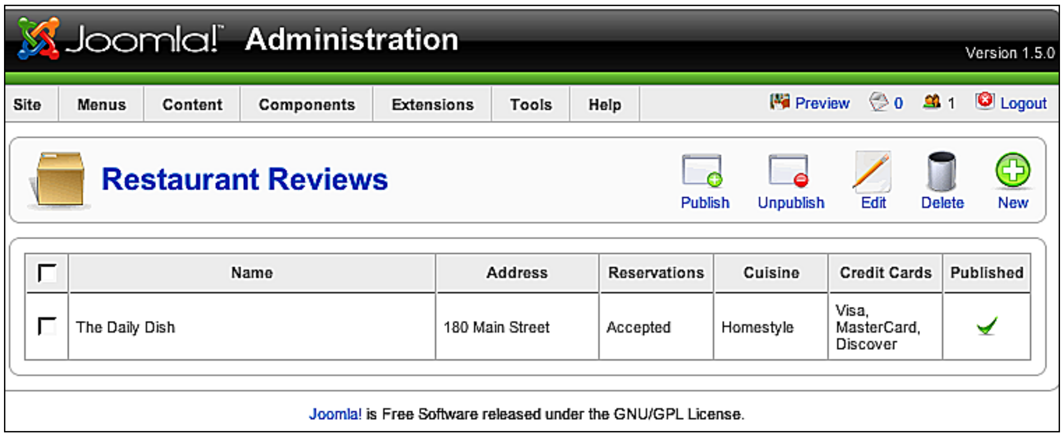
Once out of the header, we begin a loop over the rows. The variables `$i` and `$n` are initially set to `0` and the number of rows respectively; the loop runs as long as there are rows available to display. Once inside the loop, we get a reference to the current row so we can display the contents. We switch the value of `$k` back and forth between `0` and `1`; this is used to alternate between two different CSS classes with slightly different background properties.

Several of the member variables are output directly, but a couple of the columns warrant special treatment. Using the `JHTML::('grid.id')`, we can get the HTML code for a checkbox that will be recognized by the back-end JavaScript. The `JHTML::_('grid.published')` function generates an image button based on the value of the `published` member variable in the row. When it is set to `1`, we get a "check" image, while a value of `0` yields an "x" image.

Below the table, there are four hidden variables. The first one holds the value for `option` so that we are routed to the correct component. The `task` is made available so that the JavaScript in the toolbars can set it before submitting the form. When any of the checkboxes for the records are toggled, `boxchecked` is set to `1`. It is set back to `0` when all checkboxes are cleared. This aids the JavaScript in processing the list. Once the HTML output code is in place, update your `switch()` statement in `admin.reviews.php` with the following highlighted code. This will add a default case for when no task is selected:

```
switch($task)
{
  case 'add':
  editReview( $option );
    break;
  case 'save':
    saveReview( $option );
    break;
  default:
    showReviews( $option );
    break;
}
```

When you now load `http://localhost/joomla/administrator/index.php?option=com_reviews`, a screen similar to the following should appear:



☐	Name	Address	Reservations	Cuisine	Credit Cards	Published
☐	The Daily Dish	180 Main Street	Accepted	Homestyle	Visa, MasterCard, Discover	✓

Joomla! is Free Software released under the GNU/GPL License.

Editing Records

Instead of writing a whole new set of functions for editing records, we can extend the existing code. In the `admin.reviews.php` file under `editReview()` replace:
`$row =& JTable::getInstance('Review', 'Table');`
with the following highlighted code:

```
function editReview( $option )
{
  $row =& JTable::getInstance('review', 'Table');
  $cid = JRequest::getVar( 'cid', array(0), '', 'array' );
  $id = $cid[0];
  $row->load($id);
```

As we did with the `saveReview()` function, we get a `TableReview` object to handle the data for the record. We also pull in the form variable `cid`, which is an array of record IDs. Since we only want to edit one record at a time, we select the first ID in the array and load the corresponding row. While we're in the `admin.reviews.php` file, we should add a case for `edit` to the `switch()`:

```
case 'edit':
case 'add':
editReview( $option );
  break;
```

You need to provide links that the user can click to edit the individual records. In the `admin.reviews.html.php` file under `HTML_reviews::showReviews()`, replace the function to display the name and add the first two bits of highlighted code as seen below:

```
jimport('joomla.filter.output');
$k = 0;
for ($i=0, $n=count( $rows ); $i < $n; $i++)
{
$row = &$rows[$i];
$checked = JHTML::_('grid.id', $i, $row->id );
$published = JHTML::_('grid.published', $row, $i );
$link = JOutputFilter::ampReplace( 'index.php?option=' .
                $option . '&task=edit&cid[]='. $row->id );
?>
<tr class="<?php echo "row$k"; ?>">
  <td>
    <?php echo $checked; ?>
  </td>
  <td>
    <a href="<?php echo $link; ?>">
    <?php echo $row->name; ?></a>
  </td>
  <td>
    <?php echo $row->address; ?>
  </td>
  <td>
    <?php echo $row->reservations; ?>
  </td>
  <td>
    <?php echo $row->cuisine; ?>
  </td>
  <td>
    <?php echo $row->credit_cards; ?>
  </td>
  <td align="center">
    <?php echo $published;?>
  </td>
```

To adhere to XHTML compliance, we need to make sure ampersands are represented by the code `&amp;`. We do this using the `ampReplace()` function. It is a member of the `JOutputFilter` class, which is loaded with the call to `jimport('joomla.filter.output')`. Joomla! has many different libraries for things such as XML processing and RSS output. Instead of loading the full set of libraries each time Joomla! loads, we use `jimport()` to load the code only where it is needed.

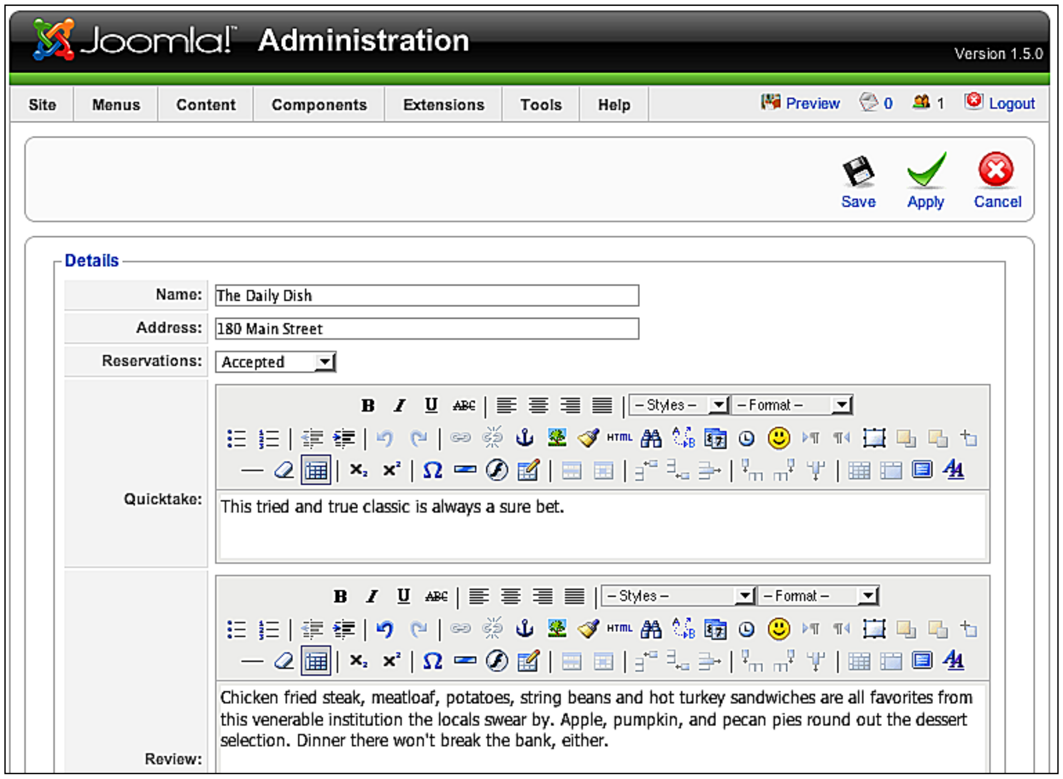
You will also have to update the toolbar code. First, go to the `switch()` in the `toolbar.reviews.php` file and check for the `'edit'` case just above `'add'`:

```
case 'edit':
case 'add':
  TOOLBAR_reviews::_NEW();
  break;
```

Now that the edit function is built, we can add an edit button that will allow you to alternatively check boxes rather than click the links. Go to the `toolbar.reviews.html.php` file and check for the following line in `TOOLBAR_reviews::_DEFAULT()`:

```
JToolBarHelper::unpublishList();
JToolBarHelper::editList();
JToolBarHelper::deleteList();
```

Save all your files and then refresh the page `http://localhost/joomla/administrator/index.php?option=com_reviews`. The record should now appear with a link. Click this link and you should get a screen that appears similar to the following:



Name:	The Daily Dish
Address:	180 Main Street
Reservations:	Accepted
Quicktake:	This tried and true classic is always a sure bet.
Review:	Chicken fried steak, meatloaf, potatoes, string beans and hot turkey sandwiches are all favorites from this venerable institution the locals swear by. Apple, pumpkin, and pecan pies round out the dessert selection. Dinner there won't break the bank, either.

You may have noticed the **Apply** button in the toolbar on the edit screen. This is intended to allow people to save their progress and continue editing the record. In order to make this button function as intended, we will have to make two changes in the `admin.reviews.php` file. Modify the `switch()` as follows:

```
case 'apply':
case 'save':
saveReview( $option, $task );
  break;
```

Add the following highlighted parameter to the function definition:

```
function saveReview( $option, $task )
```

Then change the last line of `saveReview()` to the following code that checks the current `$task`:

```
switch ($task)
{
  case 'apply':
    $msg = 'Changes to Review saved';
    $link = 'index.php?option=' . $option .
       '&task=edit&cid[]='. $row->id;
    break;
  case 'save':
  default:
    $msg = 'Review Saved';
    $link = 'index.php?option=' . $option;
    break;
}
$mainframe->redirect($link, $msg);
```

Deleting Records

Adding the delete functionality is relatively simpler. Add the following case to the `switch()` in the `admin.reviews.php` file:

```
case 'remove':
removeReviews( $option );
  break;
```

Also add the `removeReviews()` function:

```
function removeReviews( $option )
{
  global $mainframe;
  $cid = JRequest::getVar( 'cid', array(), '', 'array' );
  $db =& JFactory::getDBO();
  if(count($cid))
  {
    $cids = implode( ',', $cid );
    $query = "DELETE FROM #__reviews WHERE id IN ( $cids )";
    $db->setQuery( $query );
    if (!$db->query())
    {
      echo "<script> alert('".$db->getErrorMsg()."');
      window.history.go(-1); </script>\n";
    }
  }
  $mainframe->redirect( 'index.php?option=' . $option );
}
```

We extract the `cid` form variable again and check to see if there are any `ids` in that array. If there are, we build a string that separates individual `ids` with commas, and then use this string to build a delete query. Unless there is an error while executing the query, we redirect the user back to the list screen.

Summary

We now have a fully functional back end for entering our restaurant reviews in the back end of Joomla!. We've saved ourselves from writing routine SQL statements by extending the `JTable` class. The HTML output class is now in place and it generates add, edit, and list screens for the back end. These screens take advantage of back-end JavaScript to interact with the toolbar.

Functions have been added to work with saving, editing, and deleting records. We call these functions by switching on the `task` variable. We can now get someone to start doing data entry while we build the rest of the component.

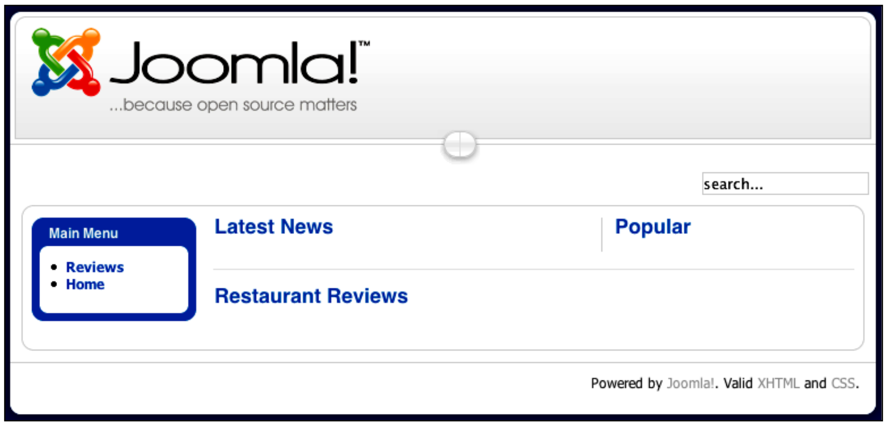
4
Front-End Development

Now that the reviewers have added some data in the back end, they're anxious to see how their reviews will appear to the visitors. While we're still working on the back end, we will learn the following about the front-end portion visible to the outside world:

- Listing the reviews
- Generating search-engine friendly links
- Displaying a review
- Adding comments
- Displaying comments

Listing the Reviews

In Chapter 2 (refer to the section *Executing the Component*) when we follow the link `http://localhost/joomla/index.php?option=com_reviews`, we get the following screen:



We will fill this screen with a list containing links pointing to the individual reviews that we had added to the database so that when visitors load the site they can navigate through the reviews.

Start by going to the `/components/com_reviews` directory and insert the following code into the `reviews.php` file:

```
jimport('joomla.application.helper');
require_once( JApplicationHelper::getPath( 'html' ) );
JTable::addIncludePath(JPATH_ADMINISTRATOR.DS.
                 'components'.DS.$option.DS.'tables');
switch($task)
{
  default:
    showPublishedReviews($option);
    break;
}
function showPublishedReviews($option)
{
  $db =& JFactory::getDBO();
  $query = "SELECT * FROM #__reviews WHERE
          published = '1' ORDER BY review_date DESC";
  $db->setQuery( $query );
  $rows = $db->loadObjectList();
  if ($db->getErrorNum())
  {
    echo $db->stderr();
    return false;
  }
  HTML_reviews::showReviews($rows, $option);
}
```

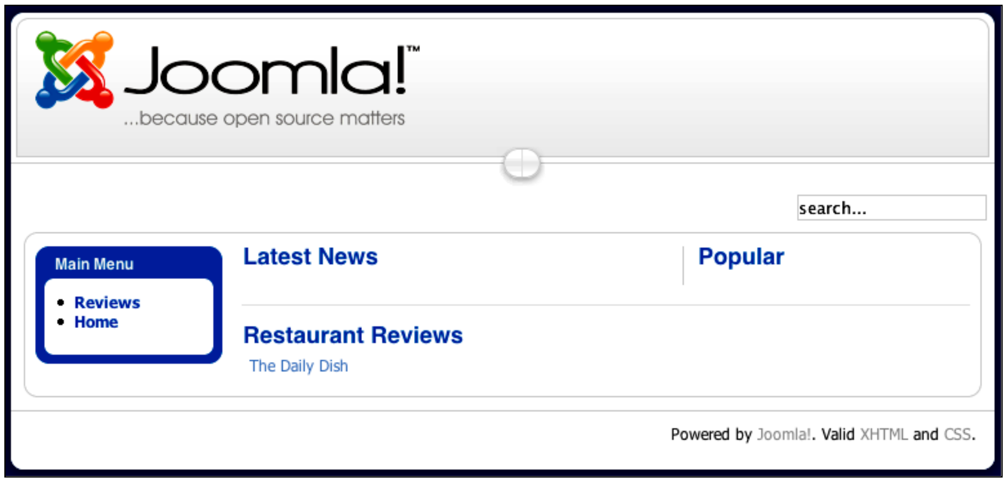
In a similar way to the back end, the code `require_once( JApplicationHelper::getPath( 'html' ) );` includes in the `reviews.html.php` file. We pass `JPATH_ADMINISTRATOR.DS.'components'.DS.$option.DS.'tables'` into `Jtable::addIncludePath();` to pull in the `table` class we wrote for the administrator portion in the previous chapter. Finally, the `switch()` function is set with a default case, which calls a function to display all the published reviews. The query in this function ensures that only the published reviews are loaded and that these are reverse-chronologically ordered by the review date.

Before we reload the page, we need to add the `HTML_reviews` class for the front end. In the `/components/com_reviews` folder, create the file `reviews.html.php`:

```
<?php
class HTML_reviews
{
  function showReviews($rows, $option)
  {
    ?><table><?php
    foreach($rows as $row)
    {
      $link = 'index.php?option=' .
                         $option . '&id=' . $row->id .  '&task=view';
      echo
  '<tr>
    <td>
      <a href="' . $link . '">' . $row->name . '</a>
    </td>
  </tr>';
    }
    ?></table><?php
  }
}
?>
```

The previous code starts by defining the `HTML_reviews` class. All our output functions for the front end will be enclosed within it. The `showReviews()` function takes a set of database result object rows and the current component name. After starting a table, the function loops through the database results and adds a link for each row.

Save all your files and hit refresh in your browser. You should now see a listing of all the reviews in the system:



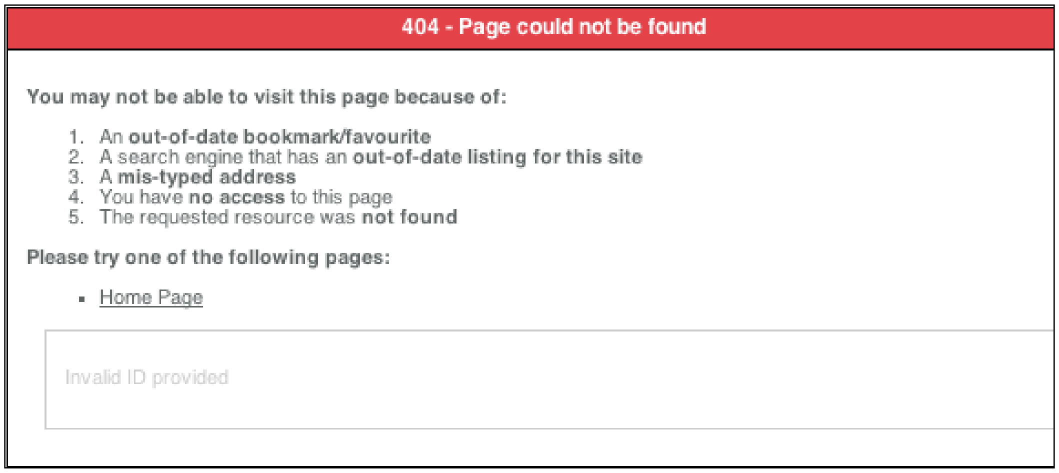
Displaying a Review

If you were to click on any of the links at the moment, you would simply see the same screen again as we have not yet coded anything to handle the `view` task. For this, add the following function to the `reviews.php` file:

```
function viewReview($option)
{
  $id = JRequest::getVar('id', 0);
  $row =& JTable::getInstance('review', 'Table');
  $row->load($id);
  if(!$row->published)
  {
    JError::raiseError( 404, JText::_( 'Invalid
                                    ID provided' ) );
  }
  HTML_reviews::showReview($row, $option);
}
```

First, we pull the `id` desired from the request by using `getVar()`, which checks the variable for different types of attacks. Externally supplied data must be handled cautiously, especially when dealing with publicly accessible websites. Consistent use of `getVar()` in our code will provide a reasonable layer of security. If the value for `id` is missing or unsuitable, the default of `0` provided in the second parameter will be used instead.

Next, we get an instance of the table class from the back end. After loading the row corresponding to the `id`, we perform a quick check to make sure that the chosen review is published. If it isn't, we use the `raiseError()` member function of `JError` to provide a **404 - Page could not be found** message.



404 - Page could not be found

You may not be able to visit this page because of:

1. An out-of-date bookmark/favourite
2. A search engine that has an out-of-date listing for this site
3. A mis-typed address
4. You have no access to this page
5. The requested resource was not found

Please try one of the following pages:

- Home Page

Invalid ID provided

This check ensures that visitors do not type in random IDs to pull up reviews that are still in progress. Conveniently, it will also fail if the record doesn't exist.

The `viewReview()` function will do everything necessary to load a requested review, but we still need to add code to call this function. Add this highlighted case of `view` to the switch on `$task`:

```
switch($task)
{
  case 'view':
    viewReview($option);
    break;
  default:
    showPublishedReviews($option);
    break;
```

We also need to create a display function in our output class. Add the `showReview()` function to `HTML_reviews` in the `reviews.html.php` file:

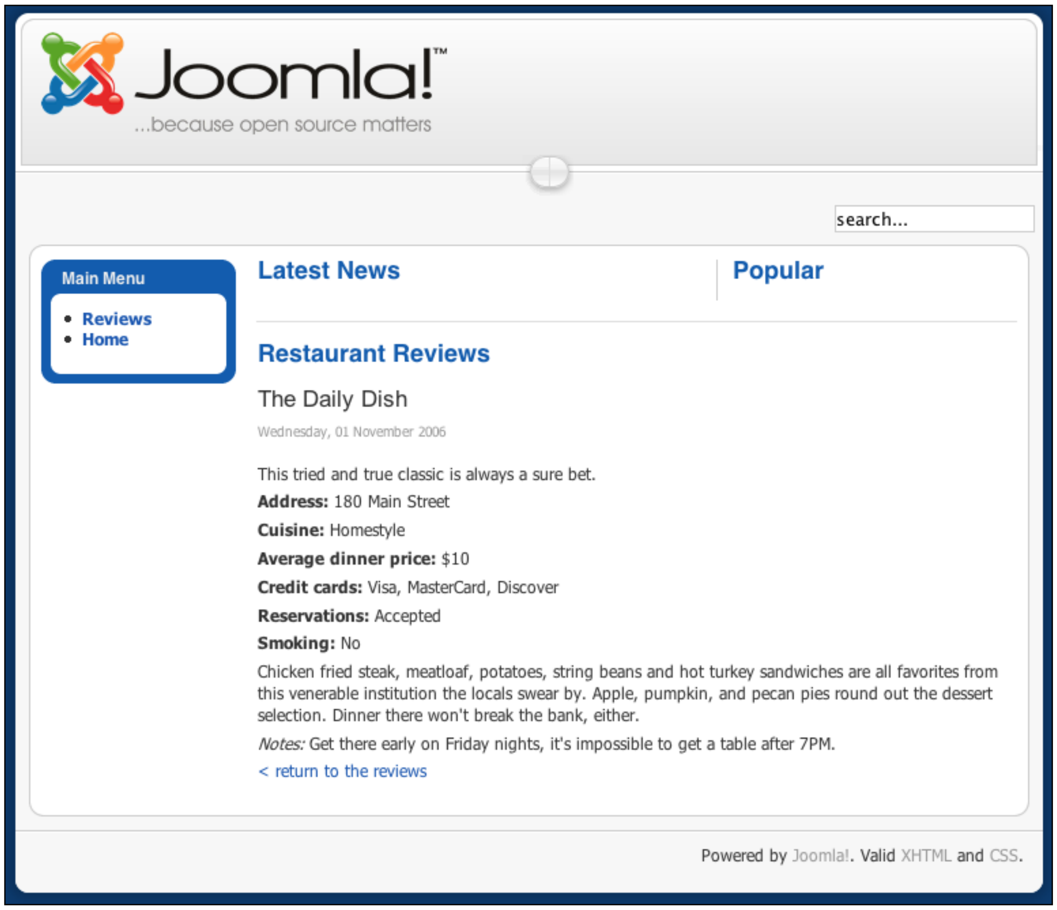
```
function showReview($row, $option)
{

  ?>
  <p class="contentheading"><?php echo $row->name; ?></p>
```

```
      <p class="createdate"><?php echo JHTML::Date
                                  ($row->review_date); ?></p>
      <p><?php echo $row->quicktake; ?></p>
      <p><strong>Address:</strong> <?php echo $row->address; ?></p>
      <p><strong>Cuisine:</strong> <?php echo $row->cuisine; ?></p>
      <p><strong>Average dinner price:</strong> $<?php echo
                              $row->avg_dinner_price; ?></p>
      <p><strong>Credit cards:</strong> <?php echo
                                  $row->credit_cards; ?></p>
      <p><strong>Reservations:</strong> <?php echo
                                  $row->reservations; ?></p>
      <p><strong>Smoking:</strong> <?php
        if($row->smoking == 0)
        {
          echo "No";
        }
        else
        {
          echo "Yes";
        }
      ?></p>
      <p><?php echo $row->review; ?></p>
      <p><em>Notes:</em> <?php echo $row->notes; ?></p>
      <?php $link = 'index.php?option=' . $option ; ?>
      <a href="<?php echo $link; ?>">&lt; return to the reviews</a>
      <?php
    }
```

The `showReview()` function takes a single database row as an object and the name of the component, as parameters. Most of the columns of the row are just displayed with HTML formatting; most of the logic has already occurred. The column **smoking** is tested and turned into a **Yes** or **No** appropriately. The call to the `JHTML::Date()` function formats the timestamp from the database into the locally preferred style. Style classes `contentheading` and `createdate` are used throughout Joomla! templates; by using them, our component blends in with the rest of the site. Finally, we link back to the review listing.

After saving all the files, click one of the review links again and you should see a nicely formatted page.



Restaurant Reviews

The Daily Dish

Wednesday, 01 November 2006

This tried and true classic is always a sure bet.

Address: 180 Main Street

Cuisine: Homestyle

Average dinner price: $10

Credit cards: Visa, MasterCard, Discover

Reservations: Accepted

Smoking: No

Chicken fried steak, meatloaf, potatoes, string beans and hot turkey sandwiches are all favorites from this venerable institution the locals swear by. Apple, pumpkin, and pecan pies round out the dessert selection. Dinner there won't break the bank, either.

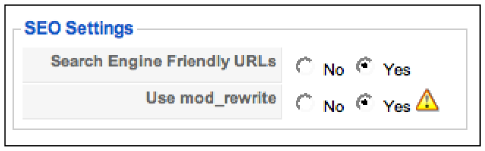
Notes: Get there early on Friday nights, it's impossible to get a table after 7PM.

< return to the reviews

Powered by Joomla!. Valid XHTML and CSS.

Generating Search-Engine Friendly Links

At this point of time the links to our reviews (`http://localhost/joomla/index.php?option=com_reviews&id=1&task=view&Itemid=1`) appear as long GET strings. Our critics mentioned that they hate seeing links like these. Also, these links are not very helpful for search engines attempting to index our site. It would be preferable to have a link like: `http://www.ourdomain.com/reviews/view/1` instead. To accomplish this, we will define a route to both generate and decode **Search-Engine Friendly (SEF)** links. Before we write any code, we will have to go to the administrator back end and enable SEF links. Go to **Site | Configuration** and make sure **Search Engine Friendly URLs** is set to **Yes**. If you're using Apache as your webserver and have `mod_rewrite` enabled, you can also set **Use mod_rewrite** to **Yes**; this will entirely remove `index.php` from your URLs. With `mod_rewrite` enabled, the **SEO Settings** portion of your **Global Configuration** screen should look like the following:


SEO Settings
Search Engine Friendly URLs: No [✓ Yes]
Use mod_rewrite: No [✓ Yes]

> If you cannot use `mod_rewrite` with your configuration, the SEF links will still be built, but will have `index.php` in the middle, for example: `http://www.yoursite.com/index.php/search/engine/friendly/link`.

Click **Save** to change the configuration. If you're using `mod_rewrite`, make sure you rename `htaccess.txt` to `.htaccess`. If you get a message saying that your configuration file is unwritable, open the `configuration.php` file in the Joomla! root and set the `$sef` member variable of `JConfig` to `1` instead of `0`.

Building URL Segments

When creating internal links while building a page in Joomla!, components and modules will call the `JRoute::_()` function. This function takes a relative link as the parameter and returns a SEF version of the link. To build this version, `JRoute::_()` first parses the relative link into an array, then removes the `option` element and adds its value as the first segment of the new URL. The function will then look for `router.php` in the component directory with the same name as `option`. If `router.php` is found, it will be included and the function beginning with the component name and ending with `BuildRoute()` will be called; in our case, `ReviewsBuildRoute()`. To create this function, go back to the `/components/com_reviews` folder and create the file `router.php`. Fill the file with the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
function ReviewsBuildRoute(&$query)
{
  $segments = array();
  if (isset($query['task']))
{
    $segments[] = $query['task'];
```

```
      unset($query['task']);
    }
    if(isset($query['id']))
    {
      $segments[] = $query['id'];
      unset($query['id']);
    }
    return $segments;
  }
  ?>
```

When `JRoute::_()` determines that the link it is processing is to a restaurant review, `ReviewsBuildRoute()` will be called and an array of the parsed URL (without the option element) will be passed in. To finish building the SEF link, we need to return an ordered array of the rest of the URL segments. First, we set `$segments` as an empty array. Next, we test the `$query` array to see if the `task` element is present. If so, we add the value of task as the first element of `$segments` and then remove `task` from `$query`. Next, we do the same process with `id`. Finally, we return `$segments` so that `JRoute::_()` can finish building the URL.

There are two methods involved in the way this function is written that are crucial to getting SEF URLs correctly built. First, the `$query` array must be passed in by reference (preceded by & in the function definition). As we build the segments, we remove the processed elements from the `$query` array. Any elements left in `$query` after our function will be processed back into the URL and appear similar to GET elements. If we do not pass in `$query` by reference, the calls to `unset()` will only effect our local copy and all of the URL elements will appear after the SEF segments.

Besides correctly handling `$query`, the ordering of the elements in `$segments` matters. Since SEF URLs do not have any way of identifying the elements that the values are intended to set, the only way we can reliably map our values is to rely on a predefined order. When we return `$segments`, `JRoute::_()` will add each element from this array to the URL, separating them by slashes. If there are any variables left in `$query`, these will be added to the end of the URL in GET string style.

Although we now have `router.php` in place with a function that will generate SEF URLs, our component's output functions are not set to use it. Open `/components/com_reviews/reviews.html.php` and check for the highlighted code in the `showReviews()` member function of `HTML_reviews`:

```
foreach($rows as $row)
{
  $link = JRoute::_('index.php?option=' . $option . '&id=' .
                      $row->id . '&task=view');
  echo '<tr><td><a href="' . $link . '">' .
                         $row->name . '</a></td></tr>';
}
```

Also check for the following highlighted code in `HTML_reviews::showReview()`:

```
<p><em>Notes:</em> <?php echo $row->notes; ?></p>
<?php $link = JRoute::_('index.php?option=' . $option); ?>
<a href="<?php echo $link; ?>">&lt; return to the reviews</a>
```

The component will now generate SEF URLs according the pattern we set in `ReviewsBuildRoute()`.

Parsing URL Segments

If you attempt to click on one of the reviews right now, you will get a message like **"Fatal error: Call to undefined function reviewsParseRoute()"**. In addition to a function generating SEF URLs for reviews, we need a function capable of decoding these URLs. Go back to `/components/com_reviews/router.php` and add the following function:

```
function ReviewsParseRoute($segments)
{
  $vars = array();
  $vars['task'] = $segments[0];
  $vars['id'] = $segments[1];
  return $vars;
}
```

Once Joomla! determines that the page request is intended for the reviews component, it will call `BuildParseRoute()` and pass in an array of the relevant URL segments. These segments are ordered the same way we set them in `ReviewsBuildRoute()`. We initialize an array `$vars` to hold the variables we return. Then we set the `task` and `id` elements of this array to the first and second elements of `$segments` respectively. Finally, we return the array, which Joomla! in turn sets as request variables. This way, the entire routing process is transparent to the rest of the code: all of the request variables you would normally expect to be present under a conventional script call will be there.

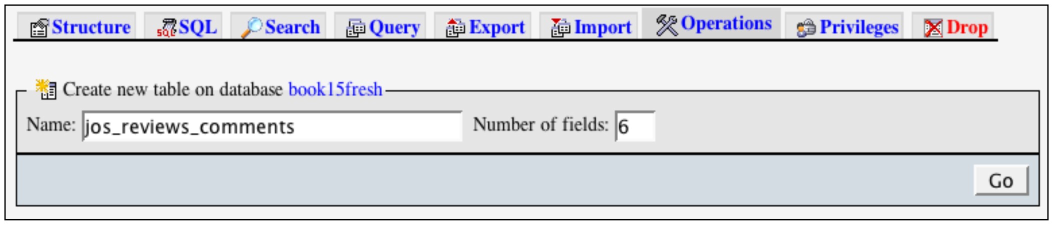
Save `router.php` and try clicking on some of the links and pay attention to the location bar in your browser. You should now notice URLs like `http://www.oursite.com/reviews/view/1` or `http://www.oursite.com/index.php/reviews/view/1`. If the URLs look like `http://www.oursite.com/component/reviews/view/1`, this just means that you followed a non-SEF URL; this will clear up as you navigate around.

Adding Comments

Most visitors will take our word for it when we say that a restaurant is great (or that it isn't). However, there may be a few who disagree. Why not give them an opportunity to leave comments about their experiences with the restaurant? We'll need a place to store them, so enter the following SQL command into your database console:

```
CREATE TABLE 'jos_reviews_comments' (
  'id' int(11) NOT NULL auto_increment,
  'review_id' int(11) NOT NULL,
  'user_id' int(11) NOT NULL,
  'full_name' varchar(50) NOT NULL,
  'comment_date' datetime NOT NULL,
  'comment_text' text NOT NULL,
  PRIMARY KEY  ('id')
)
```

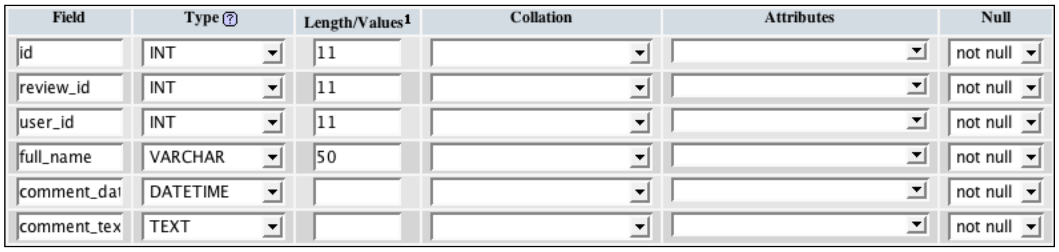
If you're using phpMyAdmin, pull up the following screen and enter **jos_reviews_comments** as the table name and **6** in the **Number of fields** section:



Create new table on database book15fresh

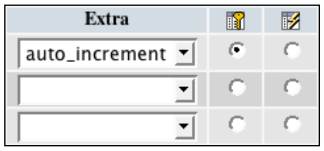
Name: jos_reviews_comments Number of fields: 6

After clicking **Go**, a grid is displayed; fill in details so that it looks like the following screen:



Field	Type	Length/Values1	Collation	Attributes	Null
id	INT	11			not null
review_id	INT	11			not null
user_id	INT	11			not null
full_name	VARCHAR	50			not null
comment_dat	DATETIME				not null
comment_tex	TEXT				not null

Be sure you make the field **id** into an automatically incremented primary key:



Extra		
auto_increment	●	○
	○	○
	○	○

We also want to add another database class to handle the basic functions. Since we already have the class for the reviews themselves in `administrator/components/com_reviews/tables`, we will add the second one here as well. Create the `comment.php` file and add the following `TableComment` class, making sure that each column in the table is represented as a member variable:

```
<?php
defined('_JEXEC') or die('Restricted access');
class TableComment extends JTable
{
  var $id = null;
  var $review_id = null;
  var $user_id = null;
  var $full_name = null;
  var $comment_date = null;
  var $comment_text = null;
  function __construct(&$db)
  {
    parent::__construct( '#__reviews_comments',
                                  'id', $db );
  }
}
?>
```

Now that we've established a place to hold the comments, a form should be added so that people can enter their comments in. Open the `reviews.html.php` file and add the following function to `HTML_reviews`:

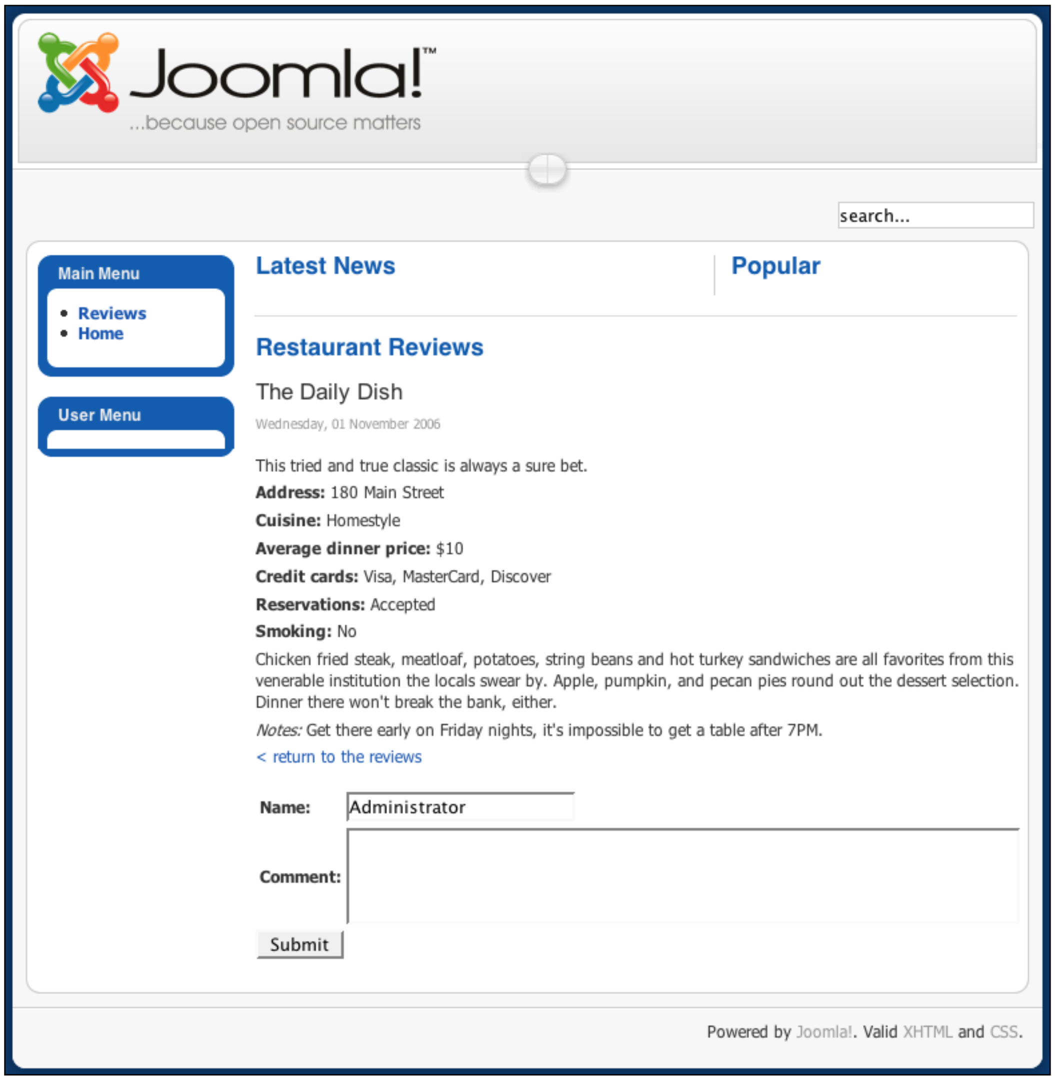
```
function showCommentForm($option, $review_id, $name)
{
  ?>
  <br /><br />
  <form action="index.php" method="post">
  <table>
    <tr>
      <td>
        <strong>Name:</strong>
      </td>
      <td>
        <input class="text_area" type="text" name="full_name"
          id="full_name" value="<?php echo $name; ?>" />
      </td>
    </tr>
    <tr>
      <td>
        <strong>Comment:</strong>
      </td>
      <td>
        <textarea class="text_area" cols="20" rows="4"
           name="comment_text" id="comment_text"
           style="width:500px"></textarea>
      </td>
    </tr>
  </table>
  <input type="hidden" name="review_id"
     value="<?php echo $review_id; ?>" />
  <input type="hidden" name="task"
     value="comment" />
  <input type="hidden" name="option"
     value="<?php echo $option; ?>" />
  <input type="submit" class="button" id="button"
     value="Submit" />
  </form>
  <?php
}
```

The `showCommentForm()` function takes the current component's name, the `id` of the review we're displaying, and a name as parameters. The **Name** is already filled in the form so that logged-in users do not have to retype it. There is a link **return to the reviews**, which routes us back to the reviews component. The `task` is set to `comment` so that the component records the comment. To make sure we attach the comment to the right review, `review_id` is set to the current one. We would like to display the form just beneath the review, so add the following highlighted code to the `viewReview()` function in the `reviews.php` file:

```
if(!$row->published)
{
  JError::raiseError( 404, JText::_( 'Invalid ID provided' ) );
}
HTML_reviews::showReview($row, $option);
$user =& JFactory::getUser();
if($user->name)
{
  $name = $user->name;
}
else
{
  $name = '';
}
HTML_reviews::showCommentForm($option, $id, $name);
```

Before calling the HTML output function, we need to get the name of the currently logged-in user (if present). The code `$user =& Jfactory::getUser();` sets `$user` as a reference to an object for the currently logged-in user. If the user's full name is set in the `name` member variable, we capture this value in `$name`, otherwise `$name` is set to a blank string.

Save all your files and reload the review. If you are logged in to the front end, your screen should look like the screenshot below. If you are not logged in, the form will be displayed, but the **Name** field will not be filled**.**



Restaurant Reviews

The Daily Dish

Wednesday, 01 November 2006

This tried and true classic is always a sure bet.

Address: 180 Main Street

Cuisine: Homestyle

Average dinner price: $10

Credit cards: Visa, MasterCard, Discover

Reservations: Accepted

Smoking: No

Chicken fried steak, meatloaf, potatoes, string beans and hot turkey sandwiches are all favorites from this venerable institution the locals swear by. Apple, pumpkin, and pecan pies round out the dessert selection. Dinner there won't break the bank, either.

Notes: Get there early on Friday nights, it's impossible to get a table after 7PM.

< return to the reviews

Name: Administrator

Comment:

Submit

Powered by Joomla!. Valid XHTML and CSS.

Before we attempt to fill in and submit the comment form, we need to add the code that will process the input and insert it into the database. Add the following highlighted code to the switch in the `reviews.php` file:

```
switch($task)
{
  case 'view':
    viewReview($option);
    break;
  case 'comment':
    addComment($option);
    break;
  default:
    showPublishedReviews($option);
    break;
}
```

Then add the `addComment()` function to the same file:

```
function addComment($option)
{
  global $mainframe;
  $row =& JTable::getInstance('comment', 'Table');
  if (!$row->bind(JRequest::get('post')))
  {
    echo "<script> alert('".$row->getError()."');
               window.history.go(-1); </script>\n";
    exit();
  }
  $row->comment_date = date( 'Y-m-d H:i:s' );
  $user =& JFactory::getUser();
  if($user->id)
  {
    $row->user_id = $user->id;
  }
  if (!$row->store())
  {
    echo "<script> alert('".$row->getError()."');
              window.history.go(-1); </script>\n";
    exit();
  }
  $mainframe->redirect('index.php?option=' .
                $option . '&id=' . $row->review_id .
               '&task=view', 'Comment Added.');
}
```

Most of this code should look familiar at this point. We get a reference to the current user object to get the user's ID and set it in our database table. At the moment, we're allowing both logged in and anonymous comments, but recording this now will give us the flexibility to track the registered users later. When the visitor is not logged in, `$user` will be empty and the `user_id` column will consequently default to `0`. Just before storing, we set `comment_date` to the current date and time. The rest of the function binds, stores, and redirects in the way the review records do in the back end.

Displaying Comments

After saving the code files, you will be able to submit the form and return to the review. However, nothing will appear to happen as we do not have the code in place to display the comments. On other websites, you will often see that the content is directly followed by the comments, which are also followed by a form for adding more comments. We will follow the same style. Add the following highlighted code to the `reviews.php` file, which will get all the comments from the database, go through them, and then output each one:

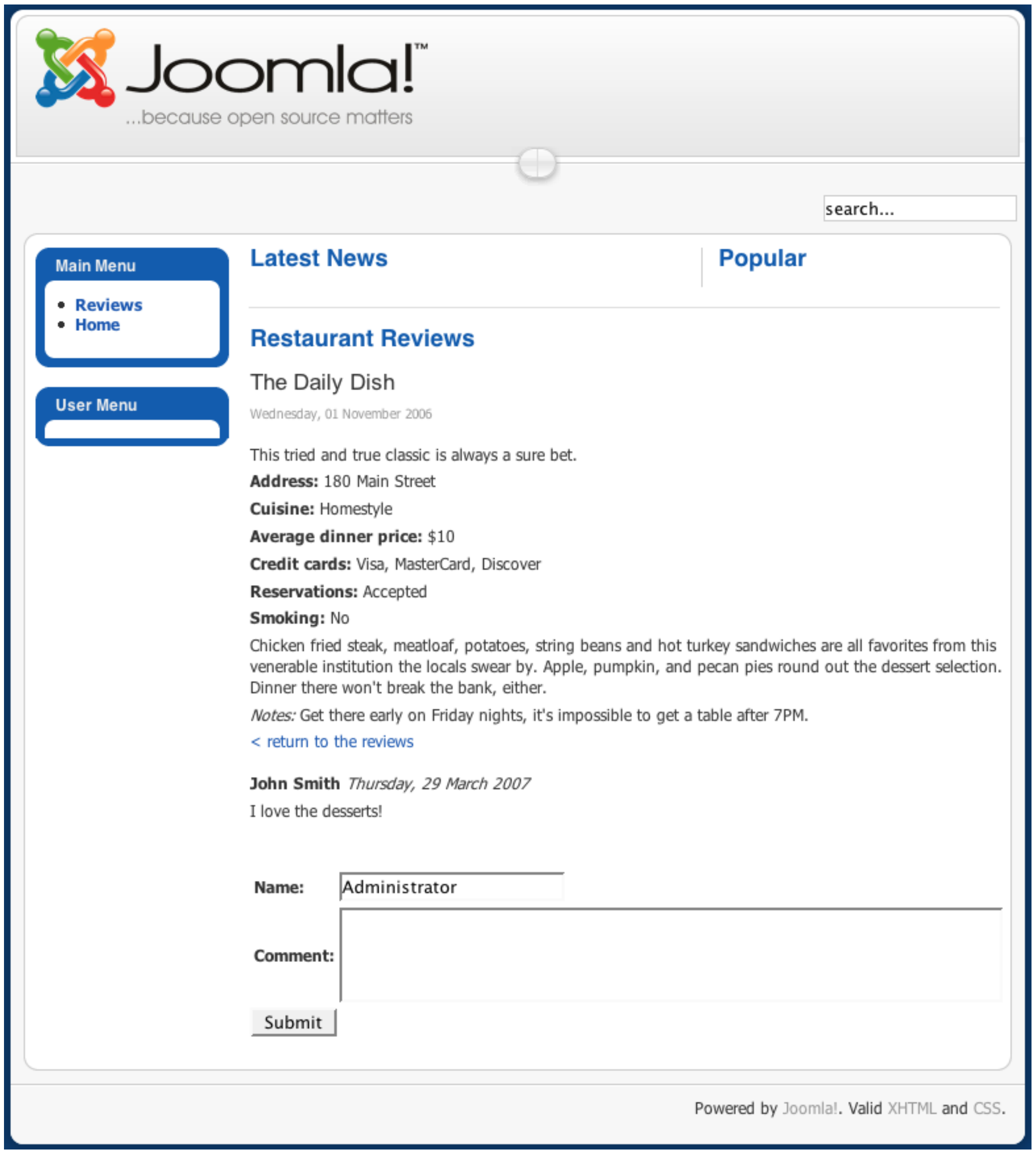
```
HTML_reviews::showReview($row, $option);
$db =& JFactory::getDBO();
$db->setQuery("SELECT * FROM #__reviews_comments
                        WHERE review_id = '$id'");
$rows = $db->loadObjectList();

foreach($rows as $row)
{
  HTML_reviews::showComment($row);
}
$user =& JFactory::getUser();
```

Also add the corresponding function in `reviews.html.php`, which outputs a single comment:

```
function showComment($row)
{
  ?>
  <br /><br />
  <p><strong><?php echo $row->full_name;
 ?></strong> <em><?php
          echo JHTML::Date($row->comment_date);
          ?></em></p>
  <p><?php echo $row->comment_text; ?></p>
  <?php
}
```

Once you've added a comment or two, refresh the review detail page and you should see a screen like the following:



Restaurant Reviews

The Daily Dish

Wednesday, 01 November 2006

This tried and true classic is always a sure bet.

Address: 180 Main Street

Cuisine: Homestyle

Average dinner price: $10

Credit cards: Visa, MasterCard, Discover

Reservations: Accepted

Smoking: No

Chicken fried steak, meatloaf, potatoes, string beans and hot turkey sandwiches are all favorites from this venerable institution the locals swear by. Apple, pumpkin, and pecan pies round out the dessert selection. Dinner there won't break the bank, either.

Notes: Get there early on Friday nights, it's impossible to get a table after 7PM.

< return to the reviews

John Smith *Thursday, 29 March 2007*

I love the desserts!

Name: Administrator

Comment:

Submit

Powered by Joomla!. Valid XHTML and CSS.

Summary

Our review site is developing successfully. Our reviewers' curiosities are being satisfied and they're beginning to get excited over the concept of being able to publish their contents consistently. We've also added some user interaction so that our visitors can agree or disagree with the reviewers and feel that they're part of the site. The links to our reviews are now more readable and ready to be crawled by search engines. This front end is a starting point from where we can add more features to make the site more enticing.

5
Module Development

We now have an efficient system for managing the reviews and taking in comments. However, visitors have to go to the component to see the reviews. The front page of our site will probably have a few articles introducing the site, but it would be nice if we could pull the content directly from the reviews and display them there as well. This is where modules can help; you can use them to fetch and display data almost anywhere on the page. In this chapter, we will cover the following topics on module development:

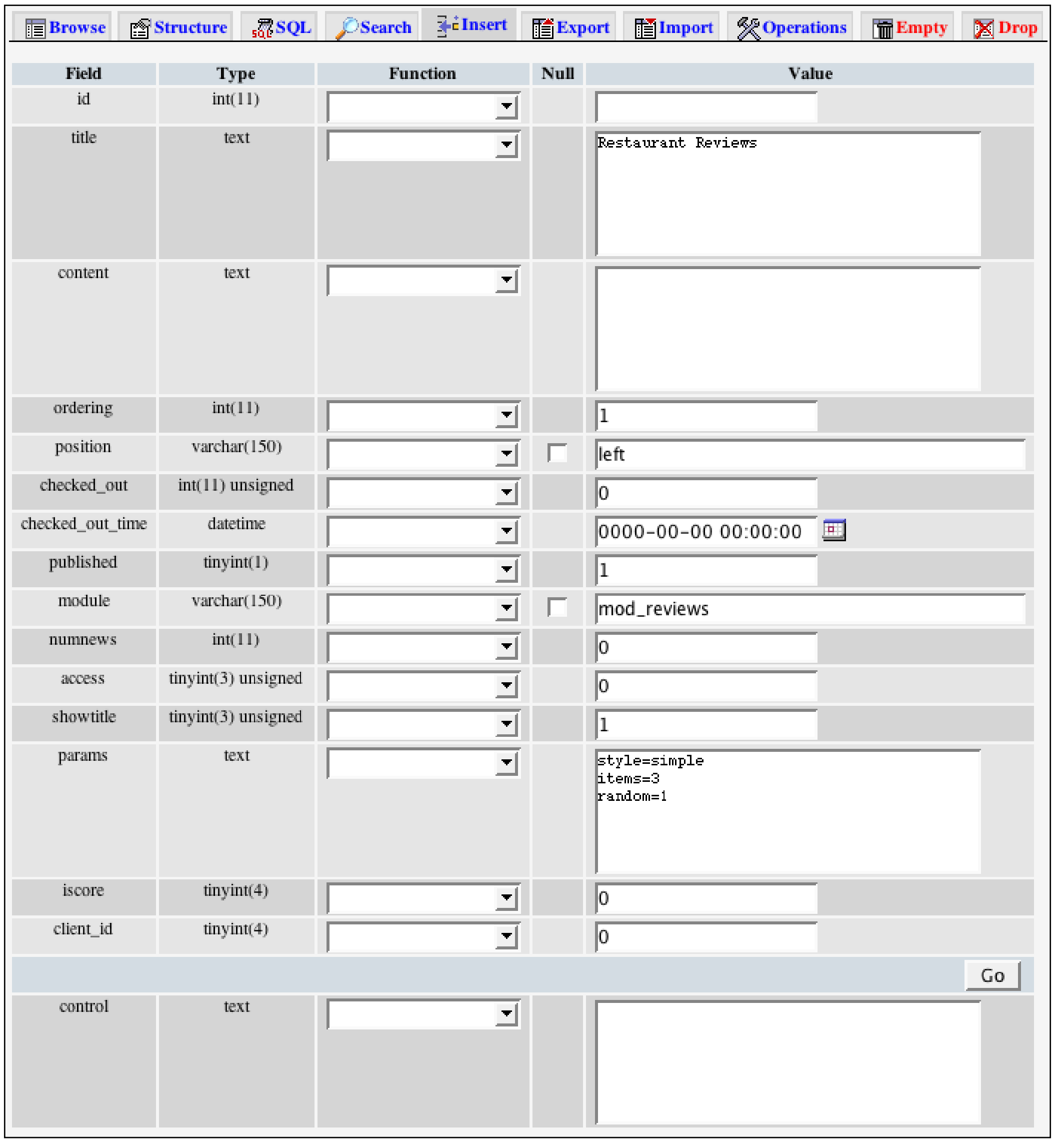
- Registering the module in the database
- Getting and setting parameters
- Centralizing data access and output using helper classes
- Selecting display options using layouts
- Displaying the latest reviews
- Displaying a random review

Registering the Module in the Database

As with the component, we will have to register the module in the database so that it can be referenced in the back end and used effectively. Entering a record into the `jos_modules` table will take care of this. Open your database console and enter the following query:

```
INSERT INTO jos_modules (title, ordering,
            position, published, module, showtitle, params)
            VALUES ('Restaurant Reviews', 1, 'left', 1,
           'mod_reviews', 1, 'style=simple\nitems=3\nrandom=1');
```

If you're using phpMyAdmin, enter the fields as in the following screen:

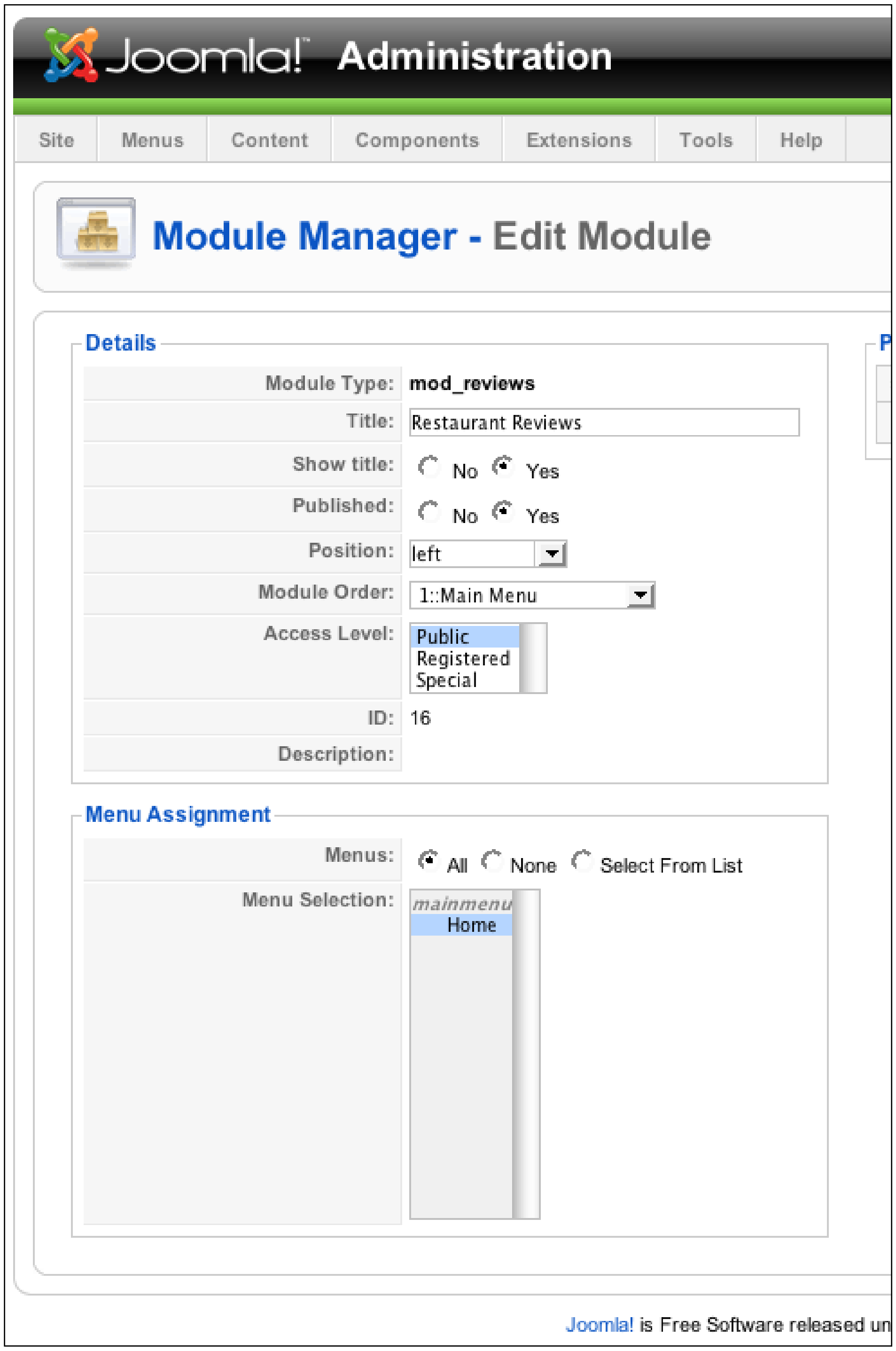


Field	Type	Function	Null	Value
id	int(11)			
title	text			Restaurant Reviews
content	text			
ordering	int(11)			1
position	varchar(150)		☐	left
checked_out	int(11) unsigned			0
checked_out_time	datetime			0000-00-00 00:00:00
published	tinyint(1)			1
module	varchar(150)		☐	mod_reviews
numnews	int(11)			0
access	tinyint(3) unsigned			0
showtitle	tinyint(3) unsigned			1

params	text			style=simple
				items=3
				random=1

iscore	tinyint(4)			0
client_id	tinyint(4)			0
control	text			

If you refresh the front end right after entering the record in `jos_modules`, you'll notice that the module doesn't appear, even though the **published** column is set to **1**. To fix this, go to **Extensions | Module Manager** in the back end and click the **Restaurants Reviews** link. Under **Menu Assignment**, select **All** and click **Save**.



Module Manager - Edit Module

Details

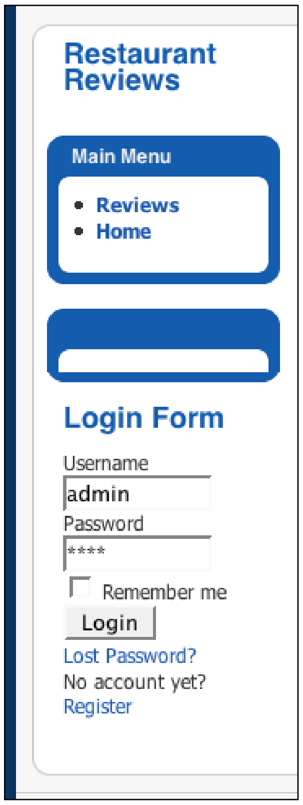
Module Type:	mod_reviews
Title:	Restaurant Reviews
Show title:	No [✓ Yes]
Published:	No [✓ Yes]
Position:	left
Module Order:	1::Main Menu
Access Level:	[✓ Public] Registered Special
ID:	16
Description:	

Menu Assignment

Menus:	[✓ All] None Select From List
Menu Selection:	*mainmenu* [✓ Home]

Joomla! is Free Software released un

In the front end, the left-hand side of your front page should look similar to the following:



Creating and Configuring a Basic Module

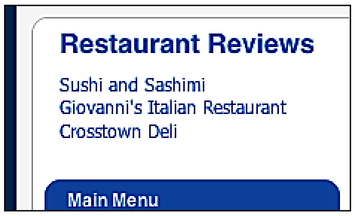
Modules are both simple and flexible. You can create a module that simply outputs static text or one that queries remote databases for things like weather reports. Although you can create rather complex modules, they're best suited for displaying data and simple forms. You will not typically use a module for complex record or session management; you can do this through a component or plug-in instead.

To create the module for our reviews, we will have to create a directory `mod_reviews` under `/modules`. We will also need to create the `mod_reviews.php` file inside `mod_reviews`. To start, we'll create a basic module that displays links to the most recent reviews. In the `mod_reviews.php` file, add the following code:

```
<?php
defined('_JEXEC') or die('Restricted access');
```

```
$items = $params->get('items', 1);
$db =& JFactory::getDBO();
$query = "SELECT id, name FROM #__reviews WHERE
                  published = '1' ORDER BY review_date DESC";
$db->setQuery( $query, 0, $items );
$rows = $db->loadObjectList();
foreach($rows as $row)
{
  echo '<a href="' . JRoute::_('index.php?option=com_reviews&id='
                   . $row->id . '&task=view') . '">' . $row->name .
                   '</a><br />';
}
?>
```

When you save the file and refresh the homepage, your module should look similar to the following:



When the module is loaded, the `$params` object is pulled into scope and can be used to get and set the parameters. When we added the row into `jos_modules`, the **params** column contained three values: one for **items** (set to **3**), one for **style** (set to **simple**), and another for **random** (set to **1**). We set `$items` to the parameter **items** using the `get()` member function, defaulting to **1** if no value exists.

If desired, you can use the member function `set($name, $value)` to override or add a parameter for your module.

After getting a database object reference, we write a query to select the **id** and **name** form `jos_reviews` and order reverse chronologically by the published date. We use the second and third parameters of `setQuery()` to generate a LIMIT clause that is automatically added to the query. This ensures that the correct syntax is used for the database type. Once the query is built, we load all the relevant database rows, go through them, and provide a link to each review.

Recruiting Some Helpers

We would like to have our module do more than display links to the reviews. It would be helpful to include a summary of the review along with each link or have the opportunity to display a review at random. However, the way we have it coded currently is not sufficient to handle different scenarios efficiently. To fix this, we will centralize the data functions into a helper class. Create the `helper.php` file in `/modules/mod_reviews` and add the following code:

```
<?php
defined('_JEXEC') or die('Restricted access');
class modReviewsHelper
{
  function getReviews(&$params)
  {
  $items = $params->get('items', 1);
    $db =& JFactory::getDBO();
    $query = "SELECT id, name, quicktake FROM #__reviews
             WHERE  published = '1' ORDER BY review_date DESC";
    $db->setQuery( $query, 0, $items );
    $rows = $db->loadObjectList();
    return $rows;
  }
  function renderReview(&$review, &$params)
  {
    $link = JRoute::_
            ("index.php?option=com_reviews&task=view&id=" .
             $review->id);
    require(JModuleHelper::getLayoutPath
             ('mod_reviews', '_review'));
  }
}
?>
```

The function `getReviews()` performs the same database actions as the original module, except that it returns the rows instead of going through them. This way, we separate the database functionality from the display logic. The **quicktake** column has been added to the query to gather the content necessary for longer review display formats.

We're going to use `renderReview()` to output single reviews. To create the link to a review, we pass `index.php?option=com_reviews&task=view&id=` and the review `id` into `Jroute::_()` to make our links search engine friendly. Finally, we use `require()` and the `getLayoutPath()` member function of `JModuleHelper` to include the `_review` template we're about to create.

Try Some Different Layouts

The helper class does not produce any output itself. Instead, `renderReview()` formats the link and then calls the `getLayoutPath()` member function in `JModuleHelper` to include a layout file named `_review`.

To create this file, make the folder `tmpl` under `/modules/mod_reviews`, create `_review.php` inside `tmpl`, and then add the following line of code:

```
<?php defined('_JEXEC') or die('Restricted access'); ?>
<a href="<?php echo $link ?>"><?php echo $review->name; ?></a><br />
```

The underscore at the beginning of `_review` is a convention to remind us that the layout is for internal use; it is not offered as a choice to the admin. In addition to this internal layout, we can create other layouts as different display options. To start, we will create one named `default`. Add the `default.php` file in `/modules/mod_reviews/tmpl` and add the following code:

```
<?php
defined('_JEXEC') or die('Restricted access');
foreach ($list as $review){
  modReviewsHelper::renderReview($review, $params);
}
?>
```

Notice that this layout cycles through a list of reviews, calling the helper class function that prepares single reviews for display, which in turn loads in the `_review` layout. Using the same method, we will also create a bulleted layout. Create the `bulleted.php` file in `/modules/mod_reviews/tmpl` and add the following code to create the links as a bulleted list:

```
<?php defined('_JEXEC') or die('Restricted access'); ?>
<ul>
<?php
foreach ($list as $review)
{
  echo "<li>";
  modReviewsHelper::renderReview($review, $params);
  echo "</li>";
}
?>
</ul>
```

The bulleted layout uses the same basic logic as the default layout; the only difference is that it wraps the results in bullets. Both options ultimately load the `_review` layout via the helper function, ensuring that the link formatting is consistent across layouts.

We now have two different display options and a helper class, but none of this code is yet accessible by the module. Open `mod_reviews.php` and replace the contents with the following code:

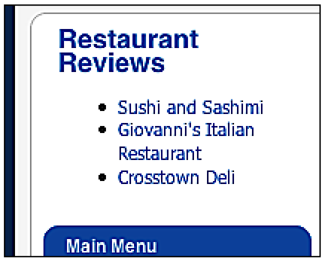
```
<?php
defined('_JEXEC') or die('Restricted access');
require(dirname(__FILE__).DS.'helper.php');
$list = modReviewsHelper::getReviews($params);
require(JModuleHelper::getLayoutPath('mod_reviews'));
?>
```

The first `require()` call pulls in the file for the helper class we just wrote. Next, we pull in a sorted set of recent reviews. Finally, we use `getLayoutPath()` to pull in a display layout. When no second parameter is specified for `getLayoutPath()`, `default` is assumed.

Save all the open files and refresh the front page in your browser. The module should look the same as in the last screenshot. Now, go back to `mod_reviews.php` and edit the call to `getLayoutPath()` so that the bulleted layout is called instead of default:

```
require(JModuleHelper::getLayoutPath('mod_reviews', 'bulleted'));
```

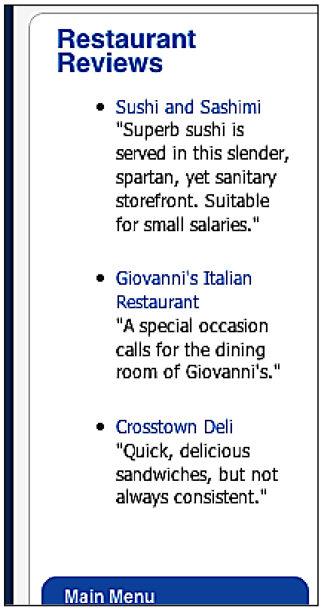
Save all the files and refresh your browser. The module should now appear similar to the following:



It would be nice if we could display a small portion of the review along with each link. Go back to `/modules/mod_reviews/tmpl/_review.php` and add the following highlighted code:

```
<a href="<?php echo $link ?>"><?php echo $review->name;
                                ?></a><br />
<p>"<?php echo $review->quicktake ?>"</p><br />
```

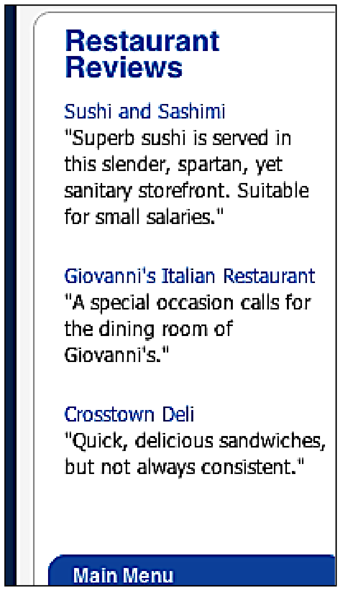
Refresh your browser. If you entered something in the **Quicktake** field in the back end when editing your reviews, you should see something like the following:



Restaurant Reviews

- Sushi and Sashimi
  "Superb sushi is served in this slender, spartan, yet sanitary storefront. Suitable for small salaries."
- Giovanni's Italian Restaurant
  "A special occasion calls for the dining room of Giovanni's."
- Crosstown Deli
  "Quick, delicious sandwiches, but not always consistent."

Go back to `mod_reviews.php` and set the second parameter of `getLayoutPath()` to `default`. After saving the file and refreshing the browser, you should see the same reviews and quotes as before, only without the bullets. While the layout is changing, the output from `_review` is staying constant.


Restaurant Reviews

Sushi and Sashimi
"Superb sushi is served in this slender, spartan, yet sanitary storefront. Suitable for small salaries."

Giovanni's Italian Restaurant
"A special occasion calls for the dining room of Giovanni's."

Crosstown Deli
"Quick, delicious sandwiches, but not always consistent."

Mixing it Up

Our module is great at highlighting the latest opinions of our diners, but our frequent visitors may want the past reviews. Let's fix that with some adjustments to the module. Replace the line in `mod_reviews.php` where the `$list` function is set with the highlighted code:

```
<?php
defined('_JEXEC') or die('Restricted access');
require(dirname(__FILE__).DS.'helper.php');
$random = $params->get('random', 0);
if($random)
{
  $list = modReviewsHelper::getRandomReview();
}
else
{
  $list = modReviewsHelper::getReviews($params);
}
```

```
require(JModuleHelper::getLayoutPath
                   ('mod_reviews', 'default'));
?>
```

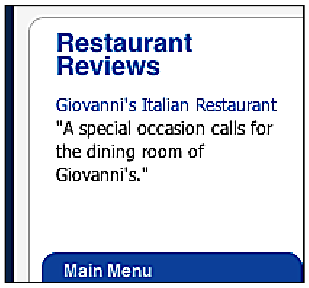
Instead of simply pulling all the reviews, we now populate `$list` based on the module's parameters. The `$params` object is automatically placed into a global scope, pre-loaded with the settings for our module. We use the `get()` member function to pull the `random` parameter into `$random`, defaulting to `0` when there is no value. Next, we test the value of `$random`. If it is non-zero, the `getRandomReview()` member function is called, which we will be adding to `modReviewsHelper` in a moment. Otherwise, we get the reviews as we did before.

Now that the parameter checking code is in place, open `helper.php` and add the following function to the `modReviewsHelper` class:

```
function getRandomReview()
{
  $db =& JFactory::getDBO();
  $query = "SELECT id, name, quicktake FROM #__reviews";
  $db->setQuery( $query );
  $rows = $db->loadObjectList();
  $i = rand(0, count($rows) - 1 );
  $row = array( $rows[$i] );
  return $row;
}
```

The function first gets a reference to the current database connection. The query is set to load the **id**, **name**, and **quicktake** columns from all rows in the **jos_reviews** table. After all the rows have been loaded into `$rows`, PHP's `rand()` function is used to get a random value between **0** and the number of rows less one, inclusive. The variable `$row` is then set to an array containing one element: the object found at the randomly selected index of `$rows`. It is necessary to wrap `$rows[$i]` in an array as our output code is expecting one.

Save the file and refresh your browser. Then refresh it repeatedly. With any luck, single reviews should appear at random.


Restaurant Reviews
Giovanni's Italian Restaurant
"A special occasion calls for the dining room of Giovanni's."
Main Menu

Summary

Now that this module is in place, we are able to draw visitors in with content we've already entered. When updates are made to the reviews, they'll automatically be reflected in the module. We've implemented a helper class to centralize some of our data access and display functions. Several different layouts have been added so that we have multiple choices for display. This module can be used anywhere on the site and will show our visitors the variety of restaurants we've reviewed.

6
Expanding the Project

Our components and modules are doing a good job of managing the reviews and taking in comments. However, there are a lot of modifications we could make to give our reviewers greater control over the display. Also, now that we have comments, we need a way of moderating them. We will make the following modifications and additions in this chapter:

- Building data models
- Migrating to views
- Switching through controllers
- Publishing controls for reviews
- Pagination for long lists
- Managing comments
- Additional toolbars

Model, View, Controller: Why?

So far, our component is doing a good job of separating the HTML output from everything else, making it relatively simpler to go back and change the layout. However, we can take this separation further to allow more flexibility. Instead of displaying the reviews in the current flat list, we could create views that would allow for columns and varying levels of detail. These options could be given to an admin who would be able to choose the desired view.

To do this, we will recode the component to follow the **Model, View, Controller (MVC)** design. Many programmers use MVC as a predictable way of controlling the logic flow in software. Models are used to define the different ways in which data can be accessed. Views generate output when given data. Controllers receive commands and route the software to the corresponding tasks and views.

In our MVC implementation, we will create data models to represent information from the database, views to display our data, and controllers that will merge the two together and handle any other task.

Building Data Models

Before we separate out our views, we need to build some data models that will get the information we will display. The front end of the component has two main screens: one that displays all the reviews and another that displays a single review. Both of these will need at least one model.

Modeling All Reviews

In the directory `/components/com_reviews`, create a folder named `models`. In this folder, create the `all.php` file and add the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
jimport( 'joomla.application.component.model' );
class ModelReviewsAll extends JModel
{
  var $_reviews = null;
  function getList()
  {
    if(!$this->_reviews)
    {
      $query = "SELECT * FROM #__reviews WHERE published = '1'";
      $this->_reviews = $this->_getList($query, 0, 0);
    }
    return $this->_reviews;
  }
}
?>
```

First, we import Joomla!'s data model libraries. Next, we declare `ModelReviewsAll` as an extension of `JModel`. Notice that we have `Model`, followed by the component name, which is followed by the model name. Following this convention makes it easier for other components to reference this model when necessary. Once inside the class, we declare `$_reviews` as a member variable; the underscore reminds us that it is a protected variable. Our single member function `getList()` checks to see if the list of reviews has been loaded. If not, we build a query to get all published reviews and send it through the `_getList()` member function of `JModel`. Once `$_reviews` has been set with the rows, we return them.

Why Are the Second and Third Parameters of _getList() Set to 0?

The second and third parameters for `_getList()` are the start and limit rows. If we wanted rows 20 through 50, we could set the second and third parameters to 20 and 30 respectively. Otherwise, `_getList()` interprets two zeros as get all rows.

Modeling Individual Reviews

In addition to modeling all published reviews, we also need a model for the individual reviews. In the `models` folder, create the `review.php` file and add the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
jimport( 'joomla.application.component.model' );
class ModelReviewsReview extends JModel
{
  var $_review = null;
  var $_comments = null;
  var $_id = null;
  function __construct()
  {
    parent::__construct();
    $id = JRequest::getVar('id', 0);
    $this->_id = $id;
  }
  function getReview()
  {
    if(!$this->_review)
    {
      $query = "SELECT * FROM #__reviews WHERE
                              id = '" . $this->_id . "'";
      $this->_db->setQuery($query);
      $this->_review = $this->_db->loadObject();
      if(!$this->_review->published)
      {
        JError::raiseError( 404, "Invalid ID provided" );
      }
    }
    return $this->_review;
  }
  function getComments()
  {
    if(!$this->_comments)
    {
      $query = "SELECT * FROM #__reviews_comments WHERE
                      review_id = '" . $this->_id . "'";
      $this->_comments = $this->_getList($query, 0, 0);
    }
    return $this->_comments;
  }
}
?>
```

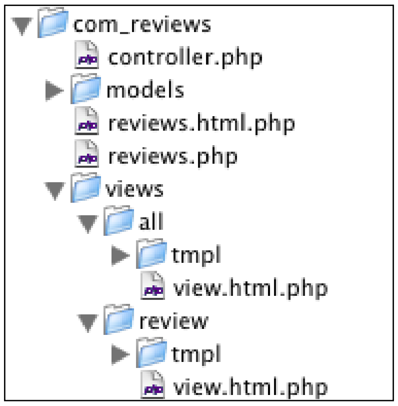
The same general concepts apply here as in the previous model. We have overridden the constructor so that we call the original one first, and then set the review `id` for the module from the request. Additionally, we have two other protected variables, which hold the information for the review and the associated comments. Instead of loading a list of records, `getReview()` loads only one row. Notice that the database object is already a protected member of the `JModel` class. If after loading the review we discover that it is unpublished, we raise a **404-Page could not be found** message. The `getComments()` function works almost identically to `ModelReviewsAll::getList()`, only querying the `#__reviews_comments` table instead.

Migrating to Views

Now that our data models are in place, we need some code that will display the information. So far, the files ending in **.html.php** have served us well for doing this task. However, the existing design is rather rigid: you include the HTML output class and call the screen you want to display. Through the use of views, we can open these screens up to the admins as choices.

Rather than keeping the entire output within a single file, we will create a separate folder for views, which will contain sub-folders for the different types of records we want to present. Within `/components/com_reviews`, create the `views` folder. At the moment, our component has functions for displaying single reviews and comments; so create two folders under `views` titled `all` and `review`. In each of these folders, create a folder for templates titled `tmpl`.

Your directory structure should look similar to the following:



Viewing All

Each view can include several templates, but needs a view object to manage these templates. To make this object for the all reviews view, create the `view.html.php` file under `components/com_reviews/views/all` and add the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
jimport('joomla.application.component.view');
class ReviewViewAll extends JView
{
  function display($tpl = null)
  {
    global $option;
    $model = &$this->getModel();
    $list = $model->getList();
    for($i = 0; $i < count($list); $i++)
    {
      $row =& $list[$i];
      $row->link = JRoute::_('index.php?option=' . $option .
                             '&id=' . $row->id  . '&view=review');
    }
    $this->assignRef('list', $list);
    parent::display($tpl);
  }
}
?>
```

After importing the core view code and declaring **ReviewViewAll** as an extension of `JView`, we create a member function `display()` that accepts the name of the template we wish to use. We get the model currently assigned to this view and use the `getList()` member function to get our set of reviews. Before heading to display the template, we go through these records and add a preformatted link using `JRoute::_()` to make them search-engine friendly. We then assign this list as a template variable and call our template. If no template name is specified, `default` is assumed. To create the default template, in `components/com_reviews/views/all/tmpl` create `default.php` and add the following code:

```
<?php defined( '_JEXEC' ) or die( 'Restricted access' ); ?>
<table>
<?php foreach($this->list as $l): ?>
<tr><td>
<a href="<?php echo $l->link; ?>"><?php echo $l->name; ?></a>
</td></tr>
<?php endforeach; ?>
</table>
```

Note that the list variable is added as a member of the current view object and therefore must be accessed through `$this`.

Viewing One

The code for displaying an individual review is similar to the code for displaying all the reviews. In components`/com_reviews/views/review` create `view.html.php` and add the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
jimport('joomla.application.component.view');
class ReviewViewReview extends JView
{
  function display($tpl = null)
  {
    global $option, $mainframe;
    $model = &$this->getModel();
    $user =& JFactory::getUser();
    $review = $model->getReview();
    $comments = $model->getComments();
    $pathway =& $mainframe->getPathWay();
    $backlink = JRoute::_('index.php?option=' .
                          $option . '&view=all' );
    $review->review_date =
                JHTML::Date($review->review_date);
```

```
        if($review->smoking == 1)
        {
          $review->smoking = "Yes";
        }
        else
        {
          $review->smoking = "No";
        }
        for($i = 0; $i < count($comments); $i++)
        {
          $row =& $comments[$i];
          $row->comment_date =
                    JHTML::Date($row->comment_date);
        }
        $pathway->addItem($review->name, '');
        $this->assignRef('review', $review);
        $this->assignRef('comments', $comments);
        $this->assignRef('backlink', $backlink);
        $this->assignRef('option', $option);
        $this->assignRef('name', $user->name);
        parent::display($tpl);
      }
    }
    ?>
```

In addition to getting variables from the model, we also pull in the object for the current user as well as the `$mainframe` object. We use the user object to get the name of the currently logged-in user and assign this to the `name` variable. On the `$review` object, we format the date and smoking columns to be user readable; we also format the dates for all the comments. From the `$mainframe` object, we pull a reference to the pathway and use the review name to add a breadcrumb. The review text, rows of comments, a link back to the list of reviews, the current component name, and the name of the currently logged-in user are all assigned to the view. Finally, we call the `display()` member function to output the review.

Creating Breadcrumbs

You can modify the pathway that commonly appears in templates by using the object returned by `$mainframe->getPathWay()`. The member function `addItem()` allows you to add a breadcrumb with the title of your choice. The first parameter is the title for the breadcrumb, while the second parameter is the URL. If you do not wish to turn the breadcrumb into a link, simply pass a blank string. This can be useful for hierarchical components where you have a series of subcategories: you can generate a link to each and then end with the title of the current record or category.

A link back to the main page is also created and assigned to the view. We finally display the chosen template after all the variables have been assigned. In `components/com_reviews/views/review/tmpl` create `default.php` and add the following code:

```
<?php defined( '_JEXEC' ) or die( 'Restricted access' ); ?>
<p class="contentheading">
  <?php echo $this->review->name; ?>
</p>
<p class="createdate">
  <?php echo $this->review->review_date; ?>
</p>
<p>
  <?php echo $this->review->quicktake; ?>
</p>
<p>
<strong>Address:</strong> <?php echo $this->review->address; ?>
</p>
<p><strong>Cuisine:</strong>
  <?php echo $this->review->cuisine; ?>
</p>
<p><strong>Average dinner price:</strong>
  $<?php echo $this->review->avg_dinner_price; ?>
</p>
<p><strong>Credit cards:</strong>
  <?php echo $this->review->credit_cards; ?>
</p>
<p><strong>Reservations:</strong>
  <?php echo $this->review->reservations; ?>
</p>
<p><strong>Smoking:</strong>
  <?php echo $this->review->smoking ?>
</p>
<p>
  <?php echo $this->review->review; ?>
</p>
<p><em>Notes:</em>
  <?php echo $this->review->notes; ?>
</p>
<a href="<?php echo $this->backlink; ?>">&lt;
           return to the reviews
</a>
<?php if(count($this->comments)) : ?>
  <br /><br />
  <?php foreach($this->comments as $comment): ?>
  <p><strong><?php echo $comment->full_name;
      ?></strong> <em><?php echo $comment->comment_date;
```

```
        ?></em></p>
    <p>
    <?php echo $comment->comment_text; ?>
  </p>
    <?php endforeach; ?>
  <?php endif; ?>
  <br /><br />
  <?php echo $this->loadTemplate('form'); ?>
```

This file is primarily composed of HTML but we use PHP calls where we want to output variables. The `name`, `review_date`, `quicktake`, `address`, `cuisine`, `credit_cards`, `reservations`, `smoking`, `review`, and `notes` are all merely echoed out. The `backlink` is embedded in an anchor tag. Before attempting to display comments, we first check to make sure that we have at least one. If so, we go through the `comments` array, displaying the `full_name`, `comment_date`, and `comment_text` for each comment. Notice that at the bottom we have a call to load another template for rendering the comments form. This allows us to keep the presentation logic for the review itself separate from the form.

However, we need to add this other template as well. In `com_reviews/views/review/tmpl` create `default_form.php` and add the following code:

```
<?php defined( '_JEXEC' ) or die( 'Restricted access' ); ?>
<form action="index.php" method="post">
<table>
<tr><td><strong>Name:</strong></td> <td><input class="text_area"
        type="text" name="full_name" id="full_name" value="<?php echo
        $this->name; ?>" /></td></tr>
<tr><td><strong>Comment:</strong></td> <td><textarea
        class="text_area" cols="20" rows="4" name="comment_text"
        id="comment_text" style="width:500px"></textarea></td></tr>
</table>
<input type="hidden" name="review_id" value="<?php
         echo $this->review->id; ?>" />
<input type="hidden" name="task" value="comment" />
<input type="hidden" name="option" value="<?php echo $option; ?>" />
<input type="submit" class="button" id="button" value="Submit" />
</form>
```

Switching Through Controllers

Our development style up to this point roughly involves testing a variable through a switch, and then calling an appropriate function. After adding a few more functions, the code will start becoming unwieldy and difficult to navigate. To head this off before it becomes a problem, we will create a controller to handle the logic flow of the component.

Create the `controller.php` file in `/components/com_reviews` and enter the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
jimport( 'joomla.application.component.controller' );
class ReviewController extends JController
{
  function display()
  {
    $document =& JFactory::getDocument();
    $viewName = JRequest::getVar('view', 'all');
    $viewType = $document->getType();
    $view = &$this->getView($viewName, $viewType);
    $model =& $this->getModel( $viewName, 'ModelReviews' );
    if (!JError::isError( $model )) {
      $view->setModel( $model, true );
    }
    $view->setLayout('default');
    $view->display();
  }
  function comment()
  {
    global $option;
    $row =& JTable::getInstance('comment', 'Table');
    if (!$row->bind(JRequest::get('post'))) {
      echo "<script> alert('".$row->getError()."');
                window.history.go(-1); </script>\n";
      exit();
    }
    $row->comment_date = date( 'Y-m-d H:i:s' );
    $user =& JFactory::getUser();
    if($user->_table->id)
    {
      $row->user_id = $user->_table->id;
    }
    if (!$row->store()) {
      echo "<script> alert('".$row->getError()."');
                window.history.go(-1); </script>\n";
      exit();
    }
    $this->setRedirect('index.php?option=' . $option .
                  '&id=' . $row->review_id . '&view=review',
                                          'Comment Added.');
    }
  }
  ?>
```

The `display()` function is called by default when the controller is executed. In here, we get the name of the requested view (defaulting to `all`). We also get the current document object by using `JFactory::getDocument()`. From this, we get the current document type with the `getType()` member function. Joomla! supports the definition of multiple document types, allowing us to create a completely different version of the output. For instance, you can have one document type for an RSS feed and another for viewing a normal HTML page. By default, `getType()` will return HTML.

Next, we get a view object by passing the view name and type into the controller's `getView()` member function. We also select a model for our view, make sure it exists, and then assign it to the view. Finally, we assign the desired layout using the view's `setLayout()` function, then tell the view to display.

The `comment()` function is essentially the same as the original `saveComment()` function from `reviews.php`, only we've used the member function `setRedirect()` to set a URL to ultimately return to after all of the processing is finished.

We finally have all the data models, views, and the controller necessary to run the component as a model-view-controller style application. All that's left is the code to execute the controller. Since Joomla! first executes `reviews.php` when the component is run, we will have to modify the code in it to use the controller instead of using the old switching system. Open `/components/com_reviews/reviews.php` and replace the existing code with the following:

```
<?php
defined('_JEXEC') or die('Restricted access');
require_once( JPATH_COMPONENT.DS.'controller.php' );
JTable::addIncludePath(JPATH_ADMINISTRATOR.DS.'components'.DS.
                         'com_reviews'.DS.'tables');
echo '<div class="componentheading">Restaurant Reviews</div>';
$controller = new ReviewController();
$controller->execute( JRequest::getVar( 'task' ) );
$controller->redirect();
?>
```

Since we're now using the controller to organize our logic flow, we use `require_once( JPATH_COMPONENT.DS.'controller.php' )` to pull in the file where we placed it. The constant `JPATH_ COMPONENT` is automatically set to the absolute path of our component's front-end directory. In addition to the controller, we want to pull in the database table classes as we did in the back end. Passing `JPATH_ADMINISTRATOR.DS.'components'.DS.'com_reviews'.DS.'tables'` into `JTable:: addIncludePath()` accomplishes this. Note that although we're building the front-end portion of the component, we're getting the table classes from the back-end directory.

After creating a new object of the `ReviewController` class, we call the `execute()` member function, passing in the task requested by the user. If no task is defined, the `display()` member function will be called. Otherwise, the name of the task will be matched to a member function of the component. This way, you can add tasks to the controller without also adding them to a lengthy `switch()` statement.

What If I Don't Want to Match 'task' to a Member Function with the Same Name?

If you ever need to override this behavior, you can call the `registerTask()` member function before calling `execute()` to add specific tasks to specific functions. The function `registerTask()` takes two parameters: the first is the value of task while the second is the name of the member function you wish to call. This is particularly useful when you want to have several tasks call the same function.

Updating Links and Routes

With the changes to our component's architecture, we need to make some changes to code we wrote earlier in our module. The view review must be specified in the generated links. In `/modules/mod_reviews/helper.php`, replace the line where `$link` is set with this code:

```
$link = JRoute::_("index.php?option=com_reviews&view=review&id=" .
                                                       $review->id);
```

Next, we need to update the router to use the view variable when building and parsing links. Open `/components/com_reviews/router.php` and change the highlighted code in `ReviewsBuildRoute()`, replacing the code that previously processed task:

```
function ReviewsBuildRoute(&$query)
{
  $segments = array();
  if (isset($query['view'])) {
   $segments[] = $query['view'];
    unset($query['view']);
  }
  if(isset($query['id']))
```

```
      {
        $segments[] = $query['id'];
        unset($query['id']);
      }
      return $segments;
    }
```

Since we are no longer using `task` in our navigational links, we no longer need to include it when building or parsing a SEF link. For updating `ReviewsBuildRoute()`, this was a simple change from `task` to `view`. However, `ReviewsParseRoute()` will need a bit of rewriting because we now have a situation where we might have 0, 1, or 2 extra segments instead of just 0 or 2. The `all` view is not accompanied by an `id`, so we need to adjust the parser to count the segments before attempting to set both `view` and `id`. Replace `ReviewsParseRoute()` with the code below:

```
    function ReviewsParseRoute($segments)
    {
      $vars = array();
      $vars['view'] = $segments[0];
      if (count($segments) > 1)
    {
        $vars['id'] = $segments[1];
        }
        return $vars;
    }
```

In this version of the function, we set initialize the `$vars` array and then set `$vars['view']` to the first element of `$segments`. If there's more than one element in `$segments`, we assume the second is an `id` and we set `$vars['id']` with it. Finally, we return `$vars`. Our links to the review listing will look like `http://www.oursite.com/reviews/all`, while links to individual reviews will look like `http://www.oursite.com/reviews/review/2`.

Reorganizing the Back-End Code

Using a controller will also benefit the back-end code that we created in Chapter 3. We can reuse most of the existing code, while gaining the benefit of not maintaining a `switch()` statement. In `/administrator/components/com_reviews` create the `controller.php` file and add the following code:

```
    defined( '_JEXEC' ) or die( 'Restricted access' );
    jimport( 'joomla.application.component.controller' );
    class ReviewController extends JController
    {
      function __construct( $default = array() )
      {
```

```
    parent::__construct( $default );
    $this->registerTask( 'add' , 'edit' );
    $this->registerTask( 'apply', 'save' );
  }
  function edit()
  {
    global $option;
    $row =& JTable::getInstance('review', 'Table');
    $cid = JRequest::getVar( 'cid', array(0), '', 'array' );
    $id = $cid[0];
    $row->load($id);
    $lists = array();
    $reservations = array(
      '0' => array('value' => 'None Taken',
             'text' => 'None Taken'),
      '1' => array('value' => 'Accepted',
             'text' => 'Accepted'),
      '2' => array('value' => 'Suggested',
             'text' => 'Suggested'),
      '3' => array('value' => 'Required',
             'text' => 'Required'),
    );
    $lists['reservations'] = JHTML::_('select.genericlist',
              $reservations, 'reservations', 'class="inputbox" '. '',
              'value', 'text', $row->reservations );
    $lists['smoking'] = JHTML::_('select.booleanlist', 'smoking',
              'class="inputbox"', $row->smoking);
    $lists['published'] = JHTML::_('select.booleanlist', 'published',
              'class="inputbox"', $row->published);
    HTML_reviews::editReview($row, $lists, $option);
  }
  function save()
  {
    global $option;
    $row =& JTable::getInstance('review', 'Table');
    if (!$row->bind(JRequest::get('post')))
    {
      echo "<script> alert('".$row->getError()."');
      window.history.go(-1); </script>\n";
      exit();
    }
    $row->quicktake = JRequest::getVar( 'quicktake', '', 'post',
                                    'string', JREQUEST_ALLOWRAW );
    $row->review = JRequest::getVar( 'review', '', 'post',
                                    'string', JREQUEST_ALLOWRAW );
    if(!$row->review_date)
      $row->review_date = date( 'Y-m-d H:i:s' );
    if (!$row->store()) {
      echo "<script> alert('".$row->getError()."'); window.history.
go(-1); </script>\n";
      exit();
    }
    switch ($this->_task)
    {
      case 'apply':
        $msg = 'Changes to Review saved';
        $link = 'index.php?option=' . $option .
                '&task=edit&cid[]='. $row->id;
        break;
      case 'save':
      default:
        $msg = 'Review Saved';
        $link = 'index.php?option=' . $option;
        break;
    }
    $this->setRedirect($link, $msg);
  }
  function showReviews()
  {
    global $option;
    $db =& JFactory::getDBO();
    $query = "SELECT * FROM #__reviews";
    $db->setQuery( $query );
    $rows = $db->loadObjectList();
    if ($db->getErrorNum()) {
      echo $db->stderr();
      return false;
    }
    HTML_reviews::showReviews( $option, $rows );
  }
  function remove()
  {
    global $option;
    $cid = JRequest::getVar( 'cid', array(), '', 'array' );
    $db =& JFactory::getDBO();
   if(count($cid))
    {
      $cids = implode( ',', $cid );
      $query = "DELETE FROM #__reviews WHERE id IN ( $cids )";
      $db->setQuery( $query );
```

```
        if (!$db->query()) {
          echo "<script> alert('".$db->getErrorMsg()."'); window.
history.go(-1); </script>\n";
        }
      }
      $this->setRedirect( 'index.php?option=' . $option );
    }
  }
```

This controller overrides the constructor, registers the add task with `edit()`, and applies the changes with `save()`. When `save()` is called, it can check `$this->_task` to find which task triggered the function call and also use the information to redirect the user appropriately.

At this point, we've elected to leave the current `admin.reviews.html.php` file in place without migrating to views. The back-end views benefit less from the new view architecture than the front-end code does; the back end does not need extensive control over the output format.

We now need to change `admin.reviews.php` to use the controller. Open this file and replace the code with the following:

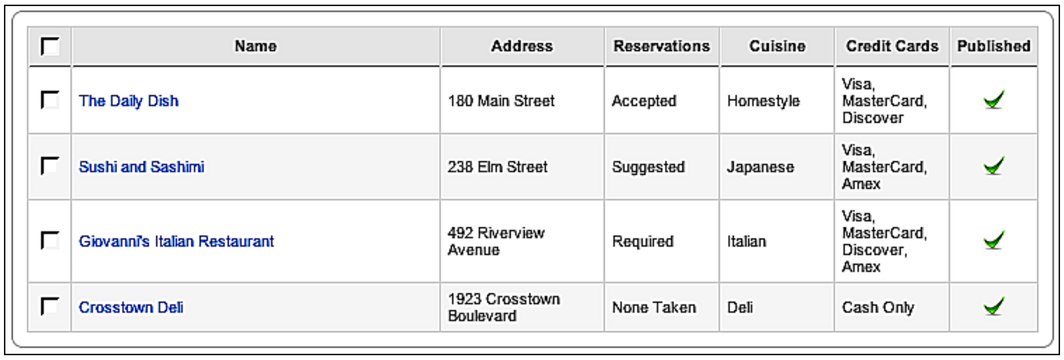
```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
require_once( JApplicationHelper::getPath( 'admin_html' ) );
require_once( JPATH_COMPONENT.DS.'controller.php' );
JTable::addIncludePath(JPATH_COMPONENT.DS.'tables');
$controller = new ReviewController(
          array('default_task' => 'showReviews') );
$controller->execute( JRequest::getVar( 'task' ) );
$controller->redirect();
?>
```

Notice that we've passed an array into the controller constructor. This array has a value for `default_task`, which allows us to tell the controller to call `showReviews()` instead of looking for `display()`. Unlike the old switch, we are not able to pass variables directly into controller member functions. If necessary, you can mimic this by adding member variables to the controller. However, most of the variables you would pass in are available through `JFactory` member functions, global objects, or from the HTTP request variables.

Once all our files are in place and a default task is assigned, call the `execute()` member function of the controller to have it perform the appropriate actions. We're passing in the `task` request variable here to tell the controller which member function to call. If no member function or registered task is found in the controller, the default task is executed. After the call to `$controller->execute()` is complete, we use the `redirect()` member function to forward the user to any URL set by `setRedirect()`. This design allows our functions to specify an ultimate destination URL without going there right away. If we need to, we can perform a "clean up" action after the controller is done executing the chosen task, and then perform the redirection.

Publishing Controls for Reviews

When we built the back-end controls for the review component, we built a list screen where the admins would be able to select an existing review for editing. At the far right-hand side of this screen, there is a column titled **Published**, which shows the current publishing status of each review.



☐	Name	Address	Reservations	Cuisine	Credit Cards	Published
☐	The Daily Dish	180 Main Street	Accepted	Homestyle	Visa, MasterCard, Discover	✓
☐	Sushi and Sashimi	238 Elm Street	Suggested	Japanese	Visa, MasterCard, Amex	✓
☐	Giovanni's Italian Restaurant	492 Riverview Avenue	Required	Italian	Visa, MasterCard, Discover, Amex	✓
☐	Crosstown Deli	1923 Crosstown Boulevard	None Taken	Deli	Cash Only	✓

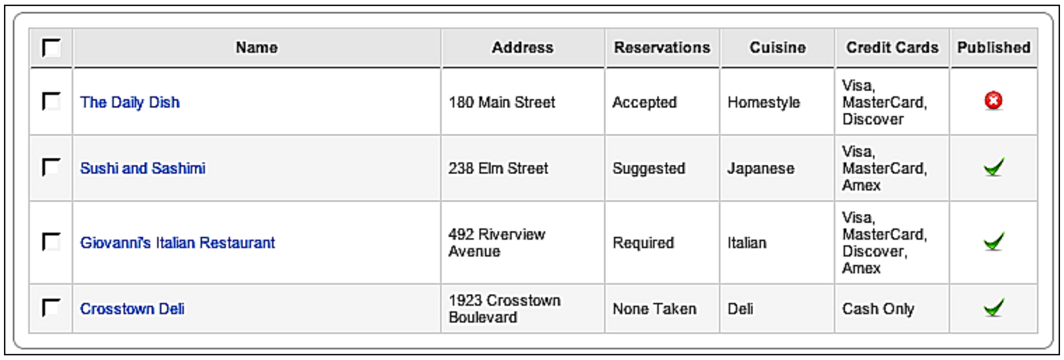
The 'check' icons in this column are actually buttons that are designed to toggle between publishing and unpublishing the reviews. If you click on one of these buttons now, the 'check' icon still remains as we have not yet added the code to make this functional. The **Publish** and **Unpublish** buttons on the toolbar are also currently non-functional. To fix this, we will add the function `publish()` to the back-end controller and register the **unpublish** task with it:

```
function __construct( $default = array() )
{
  parent::__construct( $default );
  $this->registerTask( 'add' , 'edit' );
  $this->registerTask( 'apply', 'save' );
  $this->registerTask( 'unpublish', 'publish' );
}
function publish()
{
  global $option;
```

```
    $cid = JRequest::getVar( 'cid', array(), '', 'array' );
    if( $this->_task == 'publish')
    {
      $publish = 1;
    }
    else
    {
      $publish = 0;
    }
    $reviewTable =& JTable::getInstance('review', 'Table');
    $reviewTable->publish($cid, $publish);
    $this->setRedirect( 'index.php?option=' . $option );
  }
```

Like the `edit()` function, the `publish()` function pulls in the current database object with `JFactory::getDBO()` and gets the **cid** array from our form submission. Since the publish function can handle both the **publish** and **unpublish** tasks, we check the controller's `_task` variable to discover which one is being called. Based on this, we set `$publish` to `1` or `0`. Next, `$reviewTable` is set with a reference to an instance of the `ReviewTable` class. Using the `publish()` member function of `$reviewTable`, we pass in the array of review IDs and the value we want to set for the **Published** column in the database. Finally, the controller is set to take us back to the main component screen.

After saving this code, you should be able to click on the check mark to toggle the publishing of any reviews in the back-end list.



☐	Name	Address	Reservations	Cuisine	Credit Cards	Published
☐	The Daily Dish	180 Main Street	Accepted	Homestyle	Visa, MasterCard, Discover	✗
☐	Sushi and Sashimi	238 Elm Street	Suggested	Japanese	Visa, MasterCard, Amex	✓
☐	Giovanni's Italian Restaurant	492 Riverview Avenue	Required	Italian	Visa, MasterCard, Discover, Amex	✓
☐	Crosstown Deli	1923 Crosstown Boulevard	None Taken	Deli	Cash Only	✓

Adding Pagination

Before our reviewers add too many restaurants' reviews and the list becomes rather lengthy, it would be helpful if we break this up into several screens so that they're easier to manage. Something like the search engines, which typically show ten or twenty results at a time. The functionality and interface for this is built into Joomla! and quick to add. To start, we'll add pagination to the review manager in the back end. The listings of reviews will appear across multiple pages and links will be generated to navigate between them. Open `/administrator/components/com_reviews/controller.php` and make the highlighted additions and modifications to the `showReviews()` member function:

```
function showReviews()
{
  global $option, $mainframe;
  $limit = JRequest::getVar('limit',
                $mainframe->getCfg('list_limit'));
  $limitstart = JRequest::getVar('limitstart', 0);
  $db =& JFactory::getDBO();
  $query = "SELECT count(*) FROM #__reviews";
  $db->setQuery( $query );
  $total = $db->loadResult();
  $query = "SELECT * FROM #__reviews";
  $db->setQuery( $query, $limitstart, $limit );
  $rows = $db->loadObjectList();
  if ($db->getErrorNum()) {
    echo $db->stderr();
    return false;
  }
  jimport('joomla.html.pagination');
  $pageNav = new JPagination($total, $limitstart, $limit);
  HTML_reviews::showReviews( $option, $rows, $pageNav );
}
```

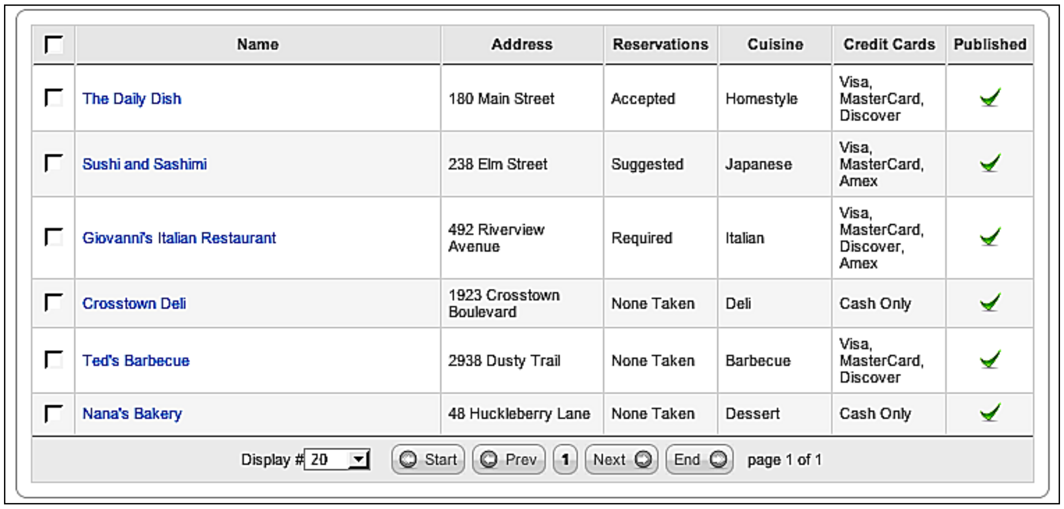
The variables `$limit` and `$limitstart` represent the maximum number of records to show and the record to start with respectively. If no limit is defined in the request, we pull it from the Joomla! configuration. To correctly calculate the number of pages to be generated, we need the total number of rows in the set; and then the `$total` variable is set to this value. When we set the query to get the reviews we wish to list, we only want to retrieve the rows we will display, so the second and third parameters of `setQuery()` are used to define this. (The database-appropriate SQL is automatically generated.)

Finally, we import the library that generates pagination HTML and get a `JPagination` class instance set with our range and total. This object is passed along to `HTML_reviews::showReviews()`, which needs to be modified to make use of the object. Open `admin.reviews.html.php` and pull up the `showReviews()` member function. Only the two small modifications highlighted below are necessary to display the pagination:

```
function showReviews( $option, &$rows, &$pageNav )
{
?>
<form action="index.php" method="post" name="adminForm">
<table class="adminlist">
  <thead>
    <tr>
      <th width="20">
        <input type="checkbox" name="toggle" value=""
          onclick="checkAll(<?php echo count( $rows ); ?>);" />
      </th>
      <th class="title">Name</th>
      <th width="15%">Address</th>
      <th width="10%">Reservations</th>
      <th width="10%">Cuisine</th>
      <th width="10%">Credit Cards</th>
      <th width="5%" nowrap="nowrap">Published</th>
    </tr>
  </thead>
  <?php
  jimport('joomla.filter.output');
  $k = 0;
  for ($i=0, $n=count( $rows ); $i < $n; $i++) {
    $row = &$rows[$i];
    $checked = JHTML::_('grid.id', $i, $row->id );
    $published = JCommonHTML::PublishedProcessing( $row, $i );
    $link = JOutputFilter::ampReplace( 'index.php?option=' .
          $option . '&task=edit&cid[]='. $row->id );
    ?>
    <tr class="<?php echo "row$k"; ?>">
      <td><?php echo $checked; ?></td>
      <td><a href="<?php echo $link; ?>"><?php echo $row->name; ?></
a></td>
      <td><?php echo $row->address; ?></td>
      <td><?php echo $row->reservations; ?></td>
      <td><?php echo $row->cuisine; ?></td>
      <td><?php echo $row->credit_cards; ?></td>
```

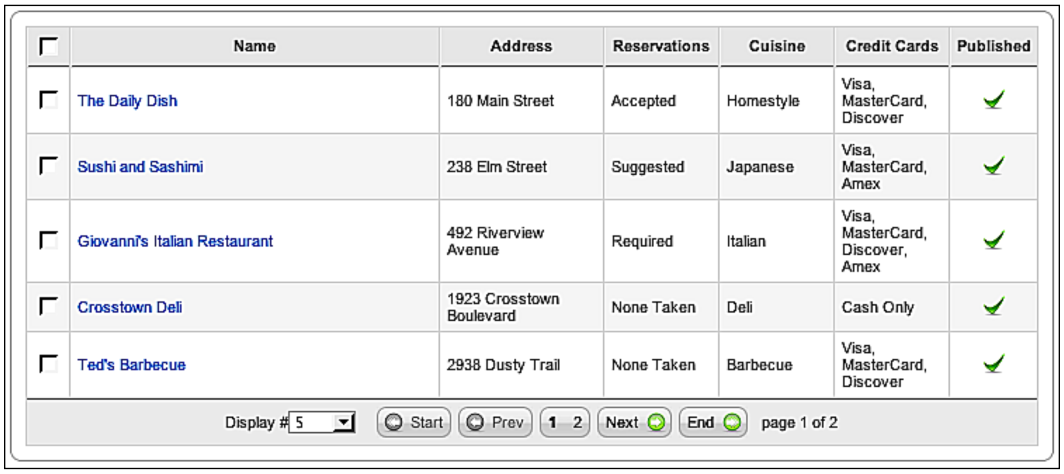
```
        <td align="center"><?php echo $published;?></td>
      </tr>
      <?php
      $k = 1 - $k;
    }
    ?>
  <tfoot>
    <td colspan="7"><?php echo $pageNav->getListFooter(); ?></td>
  </tfoot>
  </table>
  <input type="hidden" name="option" value="<?php echo $option;?>" />
  <input type="hidden" name="task" value="" />
  <input type="hidden" name="boxchecked" value="0" />
  </form>
  <?php
  }
```

The call to the `getListFooter()` member function of `$pageNav` returns HTML for links to each of the pages of review listings. The current page is highlighted, but not linked. A dropdown controlling the number of reviews to display per page is also returned. When you pull up the list in the back end, your screen should look similar to the following. You may wish to add some reviews so that you have at least six.



	Name	Address	Reservations	Cuisine	Credit Cards	Published
	The Daily Dish	180 Main Street	Accepted	Homestyle	Visa, MasterCard, Discover	✓
	Sushi and Sashimi	238 Elm Street	Suggested	Japanese	Visa, MasterCard, Amex	✓
	Giovanni's Italian Restaurant	492 Riverview Avenue	Required	Italian	Visa, MasterCard, Discover, Amex	✓
	Crosstown Deli	1923 Crosstown Boulevard	None Taken	Deli	Cash Only	✓
	Ted's Barbecue	2938 Dusty Trail	None Taken	Barbecue	Visa, MasterCard, Discover	✓
	Nana's Bakery	48 Huckleberry Lane	None Taken	Dessert	Cash Only	✓

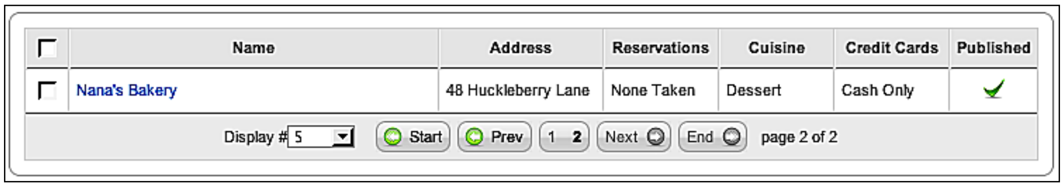
By default, your pagination is probably set at **20** or a higher number. Select **5** from the dropdown and you should be taken to a screen similar to the following:



	Name	Address	Reservations	Cuisine	Credit Cards	Published
	The Daily Dish	180 Main Street	Accepted	Homestyle	Visa, MasterCard, Discover	✓
	Sushi and Sashimi	238 Elm Street	Suggested	Japanese	Visa, MasterCard, Amex	✓
	Giovanni's Italian Restaurant	492 Riverview Avenue	Required	Italian	Visa, MasterCard, Discover, Amex	✓
	Crosstown Deli	1923 Crosstown Boulevard	None Taken	Deli	Cash Only	✓
	Ted's Barbecue	2938 Dusty Trail	None Taken	Barbecue	Visa, MasterCard, Discover	✓

Display # 5 Start Prev 1 2 Next End page 1 of 2

Clicking on either the **Next** or **End** button should yield a screen with the remaining records:



	Name	Address	Reservations	Cuisine	Credit Cards	Published
	Nana's Bakery	48 Huckleberry Lane	None Taken	Dessert	Cash Only	✓

Display # 5 Start Prev 1 2 Next End page 2 of 2

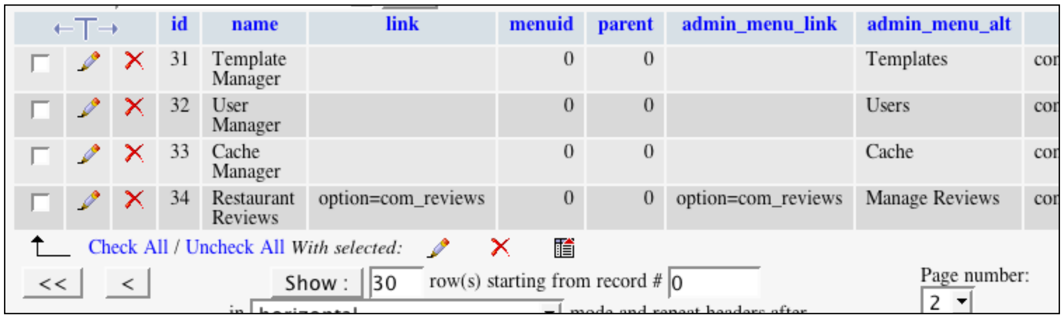
Management for Comments

We've added comments as a feature to the back end for the reviews component. Unfortunately, websites offering comments are frequently abused. We need to build a back-end manager where comments can be removed or edited. Before we actually start writing code, we need to go back to the database to add a menu item underneath the **Restaurant Reviews** link in the **Components** menu.

To create this insert query, we need to get the **id** for the current back-end component link. If you're using a command-line SQL client and your database table prefix is `jos_`, enter the following query:

```
SELECT id FROM jos_components WHERE link = 'option=com_reviews';
```

If you are using phpMyAdmin, browse the `jos_components` table until you find the row for **Restaurant Reviews** and note the value in the **id** column.



			id	name	link	menuid	parent	admin_menu_link	admin_menu_alt	
☐	✎	✗	31	Template Manager		0	0		Templates	cor
☐	✎	✗	32	User Manager		0	0		Users	cor
☐	✎	✗	33	Cache Manager		0	0		Cache	cor
☐	✎	✗	34	Restaurant Reviews	option=com_reviews	0	0	option=com_reviews	Manage Reviews	cor

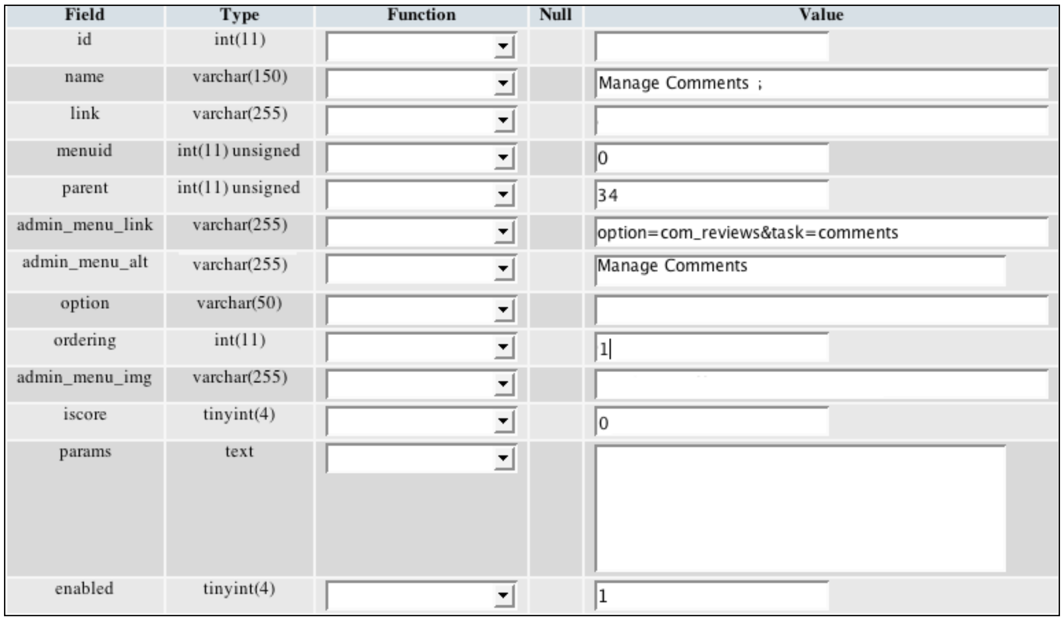
Check All / Uncheck All *With selected:*

<< < Show : 30 row(s) starting from record # 0 Page number: 2

Once you have a value for **id**, enter the following query in your SQL client, substituting **34** with the **id** in your system if necessary:

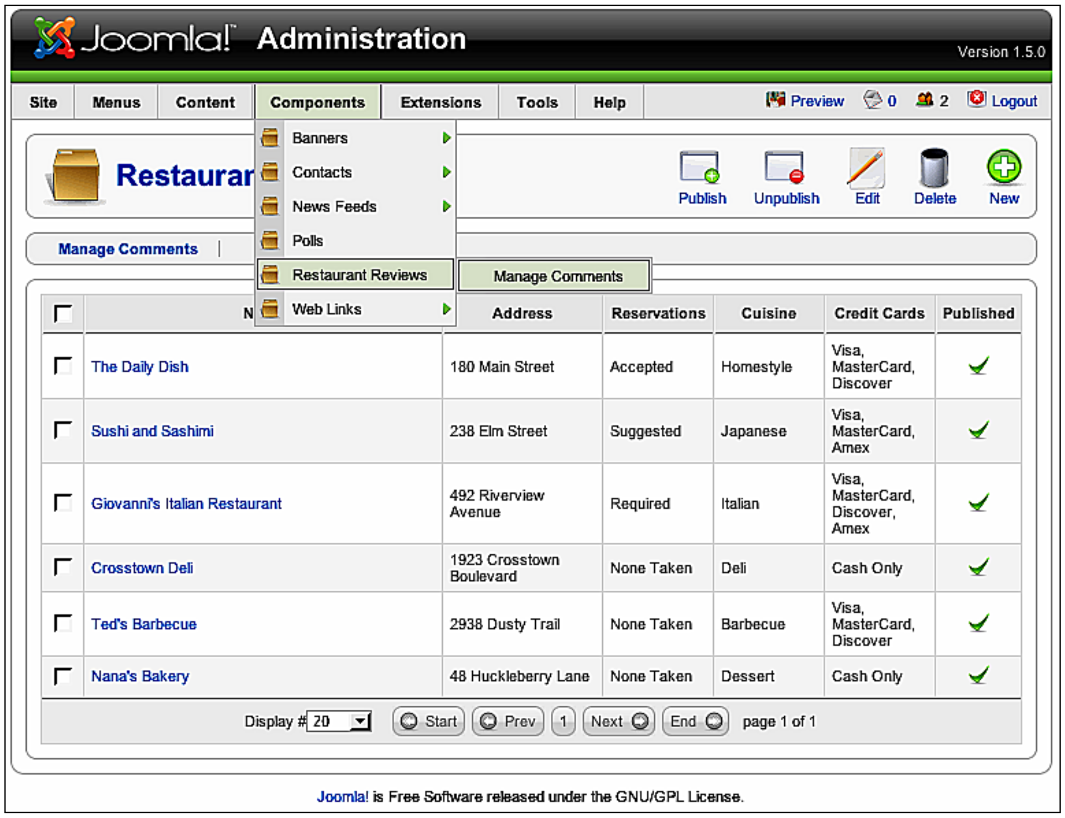
```
    INSERT INTO jos_components (name, parent, admin_menu_link,
                                         admin_menu_alt, ordering)
                             VALUES ('Manage Comments', 34,
     'option=com_reviews&task=comments', 'Manage Comments', 1);
```

If you're using phpMyAdmin, an insert screen for `jos_components` should look like the following:



Field	Type	Function	Null	Value
id	int(11)			
name	varchar(150)			Manage Comments ;
link	varchar(255)			
menuid	int(11) unsigned			0
parent	int(11) unsigned			34
admin_menu_link	varchar(255)			option=com_reviews&task=comments
admin_menu_alt	varchar(255)			Manage Comments
option	varchar(50)			
ordering	int(11)			1
admin_menu_img	varchar(255)			
iscore	tinyint(4)			0
params	text			
enabled	tinyint(4)			1

When you refresh the back end and move the cursor over the menu options, you should notice a new submenu link along with a link above the component display:



	N	Address	Reservations	Cuisine	Credit Cards	Published
	The Daily Dish	180 Main Street	Accepted	Homestyle	Visa, MasterCard, Discover	✓
	Sushi and Sashimi	238 Elm Street	Suggested	Japanese	Visa, MasterCard, Amex	✓
	Giovanni's Italian Restaurant	492 Riverview Avenue	Required	Italian	Visa, MasterCard, Discover, Amex	✓
	Crosstown Deli	1923 Crosstown Boulevard	None Taken	Deli	Cash Only	✓
	Ted's Barbecue	2938 Dusty Trail	None Taken	Barbecue	Visa, MasterCard, Discover	✓
	Nana's Bakery	48 Huckleberry Lane	None Taken	Dessert	Cash Only	✓

Now that the link is in place, let's add a screen for the link to point to. In `/administrator/components/com_reviews/controller.php`, add the following function:

```
function comments()
{
  global $option, $mainframe;
  $limit = JRequest::getVar('limit',
                 $mainframe->getCfg('list_limit'));
  $limitstart = JRequest::getVar('limitstart', 0);
  $db =& JFactory::getDBO();
  $query = "SELECT count(*) FROM #__reviews_comments";
  $db->setQuery( $query );
  $total = $db->loadResult();
  $query = "SELECT c.*, r.name FROM #__reviews_comments
```

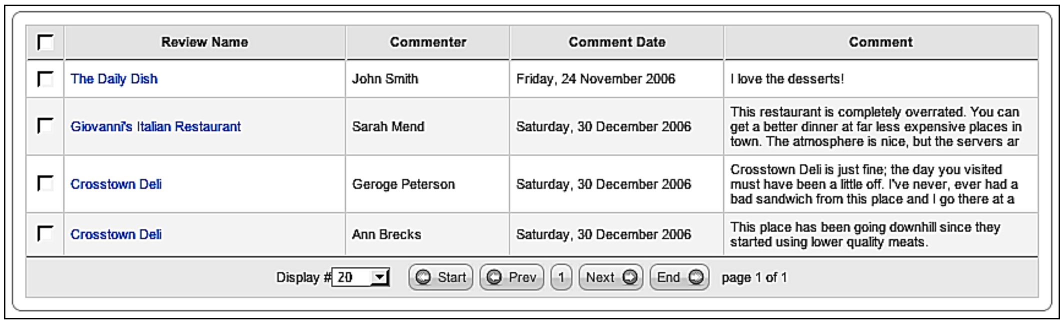
```
      AS c LEFT JOIN #__reviews AS r ON r.id = c.review_id ";
    $db->setQuery( $query, $limitstart, $limit );
    $rows = $db->loadObjectList();
    if ($db->getErrorNum())
    {
      echo $db->stderr();
      return false;
    }
    jimport('joomla.html.pagination');
    $pageNav = new JPagination($total, $limitstart, $limit);
    HTML_reviews::showComments( $option, $rows, $pageNav );
  }
```

This function is similar to the `showReviews()` function, except that we're combining the reviews table into the comments table. We end it by calling `HTML_reviews::showComments();` we will need to code this as well. Open `admin.reviews.html.php` and add the following code to the class:

```
  function showComments( $option, &$rows, &$pageNav )
  {
    ?>
    <form action="index.php" method="post" name="adminForm">
    <table class="adminlist">
      <thead>
        <tr>
          <th width="20">
          <input type="checkbox" name="toggle"
                        value="" onclick="checkAll(<?php echo
                        count( $rows ); ?>);" />
          </th>
          <th class="title">Review Name</th>
          <th width="15%">Commenter</th>
          <th width="20%">Comment Date</th>
          <th width="30%">Comment</th>
        </tr>
      </thead>
      <?php
      jimport('joomla.filter.output');
      $k = 0;
      for ($i=0, $n=count( $rows ); $i < $n; $i++) {
        $row = &$rows[$i];
        $checked = JHTML::_('grid.id', $i, $row->id );
        $link = JOutputFilter::ampReplace( 'index.php?option=' .
                 $option . '&task=editComment&cid[]='. $row->id );
        ?>
        <tr class="<?php echo "row$k"; ?>">
          <td><?php echo $checked; ?></td>
          <td><a href="<?php echo $link; ?>"><?php echo $row->name;
                                                  ?></a></td>
          <td><?php echo $row->full_name; ?></td>
          <td><?php echo JHTML::Date($row->comment_date); ?></td>
          <td><?php echo substr($row->comment_text, 0, 149); ?></td>
        </tr>
        <?php
        $k = 1 - $k;
      }
      ?>
    <tfoot>
      <td colspan="5"><?php echo $pageNav->getListFooter();
                                                  ?></td>
    </tfoot>
    </table>
    <input type="hidden" name="option"
                         value="<?php echo $option;?>" />
    <input type="hidden" name="task" value="comments" />
    <input type="hidden" name="boxchecked" value="0" />
    </form>
    <?php
  }
```

For the most part, this function is similar to the `showReviews()` counterpart. We create a form named `adminForm` so that the built-in JavaScript can interact with it. Our table header row includes a checkbox that toggles all the checkboxes on screen at once. After we output the header, we load in the output filtering classes using `jimport('joomla.filter.output')` so that we can use them to format the links to comments. Next, we set `$k` to `0` so we can alternate the table row CSS classes. This is followed by a loop through all of the comment rows. We get a checkbox and create a link for each comment. Then we output a table row with the checkbox, the link, the name of the commenter, the comment date and the comment text. We're limiting the comment text through PHP's `substr()` function so that the admins can see a portion of each comment without having them completely take over the screen. We then alternate the value of `$k` to achieve our visual effect of shading every other row. After the loop, we use the `getListFooter()` member function of `$pageNav` to provide navigation between pages of listings. Finally, we set the hidden `task` variable to **comments** and `option` to the current component name so that we land back at this screen instead of the main screen when using the pagination controls. The `boxchecked` variable is used by JavaScript to determine when any of the checkboxes are toggled on.

Now that this code is in place, follow one of the **Manage Comments** links. Your list should look similar to the following:



☐	Review Name	Commenter	Comment Date	Comment
☐	The Daily Dish	John Smith	Friday, 24 November 2006	I love the desserts!
☐	Giovanni's Italian Restaurant	Sarah Mend	Saturday, 30 December 2006	This restaurant is completely overrated. You can get a better dinner at far less expensive places in town. The atmosphere is nice, but the servers ar
☐	Crosstown Deli	Geroge Peterson	Saturday, 30 December 2006	Crosstown Deli is just fine; the day you visited must have been a little off. I've never, ever had a bad sandwich from this place and I go there at a
☐	Crosstown Deli	Ann Brecks	Saturday, 30 December 2006	This place has been going downhill since they started using lower quality meats.

We still need to build functions for editing, saving, and deleting comments. Add the following functions to the back-end controller:

```
function editComment()
{
  global $option;
  $row =& JTable::getInstance('comment', 'Table');
  $cid = JRequest::getVar( 'cid', array(0), '', 'array' );
  $id = $cid[0];
  $row->load($id);
  HTML_reviews::editComment($row, $option);
}
function saveComment()
{
  global $option;
  $row =& JTable::getInstance('comment', 'Table');
  if (!$row->bind(JRequest::get('post')))
  {
    echo "<script> alert('".$row->getError()."');
    window.history.go(-1);
    </script>\n";
    exit();
  }
  if (!$row->store()) {
    echo "<script> alert('".$row->getError()."');
    window.history.go(-1);
    </script>\n";
    exit();
  }
  $this->setRedirect('index.php?option=' . $option . '&task=comments',
'Comment changes saved');
}
```

```
function removeComment()
{
  global $option;
  $cid = JRequest::getVar( 'cid', array(), '', 'array' );
  $db =& JFactory::getDBO();
  if(count($cid))
  {
    $cids = implode( ',', $cid );
    $query = "DELETE FROM #__reviews_comments WHERE id IN ( $cids )";
    $db->setQuery( $query );
    if (!$db->query())
    {
      echo "<script> alert('".$db->getErrorMsg()."');
      window.history.go(-1);
      </script>\n";
    }
  }
  $this->setRedirect( 'index.php?option=' . $option . '&task=comments'
);
}
```

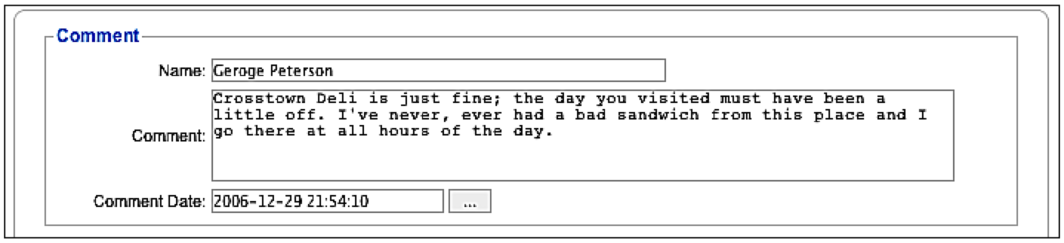
These functions are similar to the ones used for editing, saving, and removing reviews. The main difference is that we do not need to modify any of the data before storing our comments and also we need not build any HTML elements for the output class to display. The `editComment()` function calls `HTML_reviews::editComment()` which also needs to be built. Open `admin.reviews.html.php` and add the following function:

```
function editComment ($row, $option)
{
  JHTML::_('behavior.calendar');
  ?>
  <form action="index.php" method="post" name="adminForm"
                                                      id="adminForm">
    <fieldset class="adminform">
      <legend>Comment</legend>
      <table>
      <tr>
        <td width="100" align="right" class="key">
          Name:
        </td>
        <td>
          <input class="text_area" type="text" name="full_name"
          id="full_name" size="50" maxlength="250" value="<?php echo
          $row->full_name;?>" />
        </td>
      </tr>
```

```
        <tr>
          <td width="100" align="right" class="key">
            Comment:
          </td>
          <td>
            <textarea class="text_area" cols="20" rows="4"
            name="comment_text" id="comment_text"
                      style="width:500px"><?php echo $row->comment_text;
            ?></textarea>
          </td>
        </tr>
        <tr>
          <td width="100" align="right" class="key">
            Comment Date:
          </td>
          <td>
            <input class="inputbox" type="text" name="comment_date"
            id="comment_date" size="25" maxlength="19" value="<?php echo
            $row->comment_date; ?>" />
            <input type="reset" class="button" value="..."
            onclick="return showCalendar('comment_date', 'y-mm-dd');" />
          </td>
        </tr>
        </table>
      </fieldset>
      <input type="hidden" name="id" value="<?php echo $row->id; ?>" />
      <input type="hidden" name="option" value="<?php echo $option; ?>"
/>
      <input type="hidden" name="task" value="" />
    </form>
    <?php
}
```

The function begins by telling Joomla! to add the JavaScript and CSS class necessary for our pop-up date calendar. Next, we build a form named **adminForm** and create a fieldset that the admin CSS will format. We use a simple input for the name field and give it the `text_area` CSS class. The comment itself is displayed in a textarea field, also with a class of `text_area`. Finally the comment date value is held in an input field of class `inputbox`, while the button that toggles the pop-up calendar is given the class `button`. This button is set to call a the JavaScript function `showCalendar()`, which interacts with the `comment_date` field holding the date in `y-mm-dd` format. After the table and fieldset are closed, we add hidden variables for the comment `id`, the component name in `option`, and placeholder for `task` JavaScript can fill in later.

Once all of this code is in place, click on a link to one of the comments in the back end and a screen similar to the following should appear:


Comment

Name: Geroge Peterson

Comment: Crosstown Deli is just fine; the day you visited must have been a little off. I've never, ever had a bad sandwich from this place and I go there at all hours of the day.

Comment Date: 2006-12-29 21:54:10

Additional Toolbars

If you try to use any of the toolbars in place at the moment on any of the comment administration screens, you won't get the results you are expecting. This is because the buttons are still pointing to the tasks for reviews. To fix this, we will need to create some new toolbars for the comments. Open `/administrator/components/com_reviews/toolbar.reviews.html.php` and add the following class:

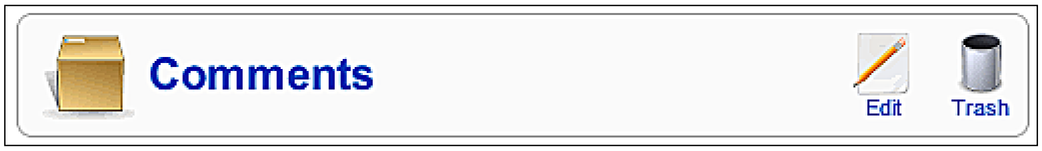
```
class TOOLBAR_reviews_comments
{
  function _EDIT()
  {
    JToolBarHelper::save('saveComment');
    JToolBarHelper::cancel('comments');
  }
  function _DEFAULT()
  {
    JToolBarHelper::title( JText::_( 'Comments' ), 'generic.png' );
    JToolBarHelper::editList('editComment');
    JToolBarHelper::deleteList('Are you sure you want to remove
                          these comments?', 'removeComment');
  }
}
```

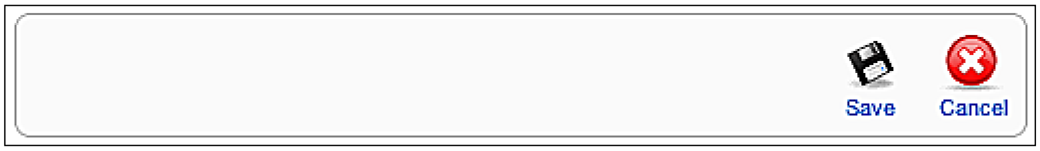
The first parameter in the calls to the `save()`, `cancel()`, and `editList()` member functions of `JToolBarHelper` overrides the default task, allowing us to redefine them with our own. The call to `title()` allows us to use the left-hand portion of the menu bar for identifying the screen, as we did with the toolbar for managing reviews. Finally, the call to `deleteList()` takes a confirmation message as the first parameter and the desired task as the second. The confirmation message is displayed when boxes are checked to ask the user to confirm before proceeding with the deletion task.

To display these toolbars, we need to modify the `switch()` in `toolbars.reviews.php` that we created in Chapter 2. The additional code is highlighted below:

```
switch($task)
{
  case 'edit':
  case 'add':
    TOOLBAR_reviews::_NEW();
    break;
  case 'comments':
  case 'saveComment':
  case 'removeComment':
    TOOLBAR_reviews_comments::_DEFAULT();
    break;
  case 'editComment':
    TOOLBAR_reviews_comments::_EDIT();
    break;
  default:
    TOOLBAR_reviews::_DEFAULT();
    break;
}
```

You should now see the toolbars below for the list and edit screens, respectively. These should work only on the comments and should not interfere with the functionality of the reviews management.





Summary

The simple component we first constructed has developed into software that can be easily expanded and updated in the future. We now have data models that represent all the information we display. Additional views allow our administrators to link to their reviews in different ways. We also have more control over publishing and comments, which will be essential as our site grows in popularity.

7
Behind the Scenes: Plug-Ins

So far, our restaurant reviews have been relatively easy to navigate and the site too is user friendly. However, our restaurant critics want us to make things simpler for them. We may periodically want to create feature articles for collections, such as *Late Night Dining Highlights* or *An All Asian Appetite*. We want to make it easier for the critics to link to the reviews within these articles and also make the reviews searchable along with the articles.

These new features can be handled through plug-ins. Joomla! provides plug-ins as a way of running pieces of code when certain events occur. Among other tasks, plug-ins can be used to start HTML editors, perform searches, format content, and log users into multiple systems at once. Plug-ins are able to interact with components and modules without modifying their source code. The plug-ins we will write will take us through the following topics:

- Database Queries
- A simple link plug-in
- An information box plug-in
- Searching reviews

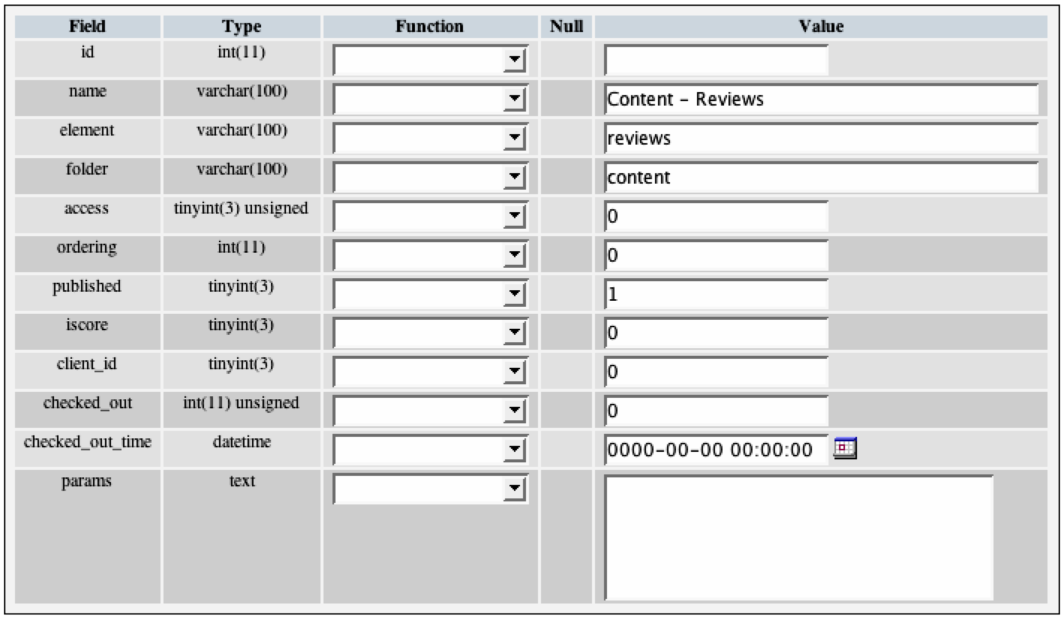
Database Queries

Before writing our code, there are some queries to be run that will register the plug-ins in the database. We will be creating three plug-ins: two plug-ins will format content and one will interact with Joomla!'s core search component. The queries will add records pointing to the folders where each can be found along with their names.

In your SQL console, enter these three queries:

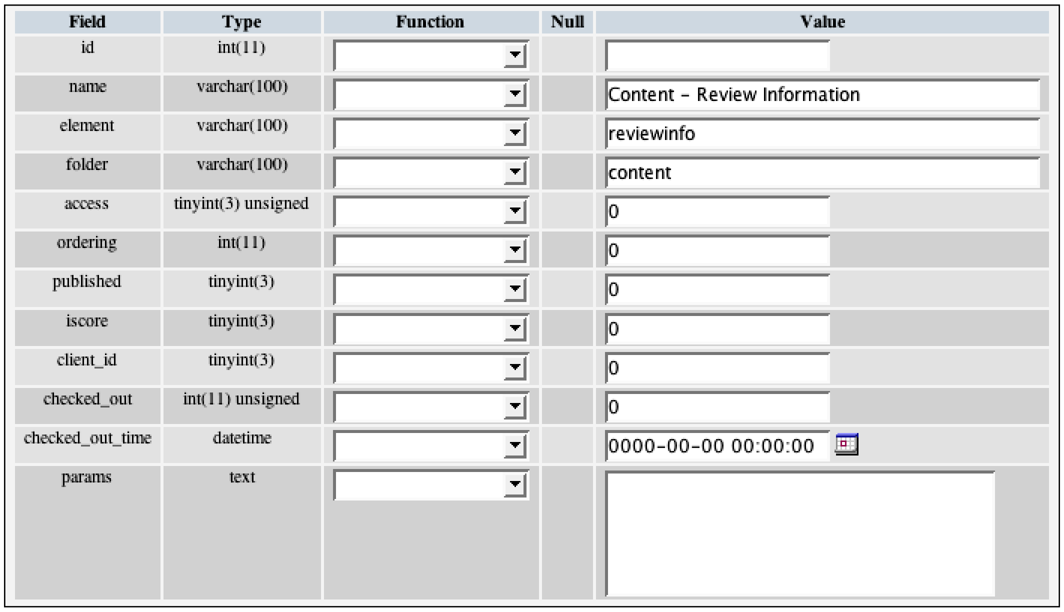
```
INSERT INTO jos_plugins (name, element, folder, published) VALUES
('Content - Reviews', 'reviews', 'content', 1);
INSERT INTO jos_plugins (name, element, folder, published) VALUES
('Content - Review Information', 'reviewinfo', 'content', 0);
INSERT INTO jos_plugins (name, element, folder, published)
      VALUES ('Search - Reviews', 'reviews', 'search', 1);
```

If you're using phpMyAdmin, pull up a screen to enter rows into `jos_plugins` and enter the information as in the following screenshot to register the **Content – Reviews** plug-in in the database:



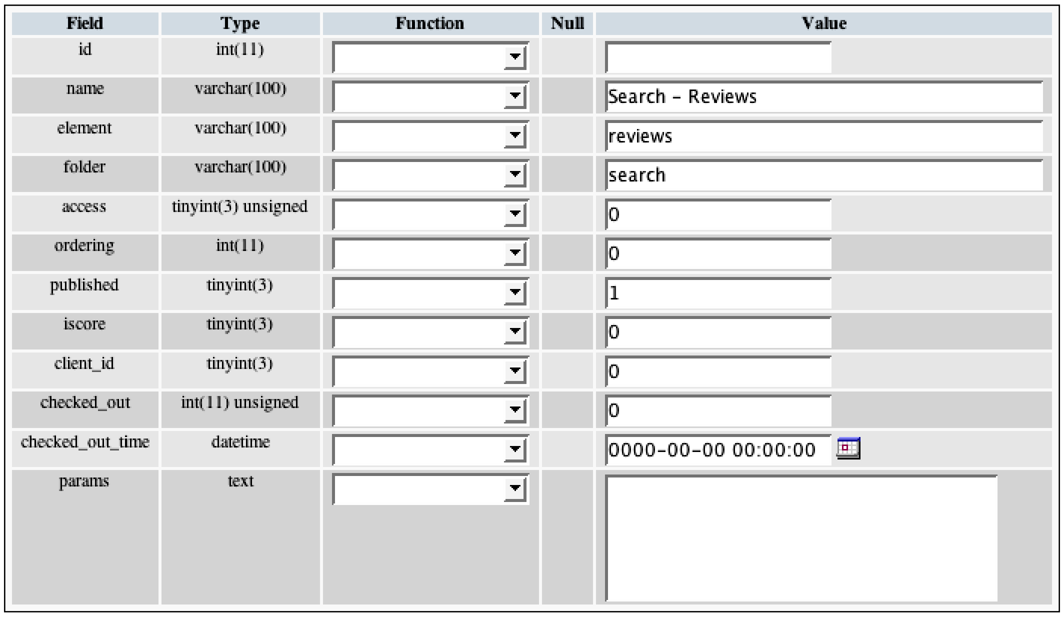
Field	Type	Function	Null	Value
id	int(11)			
name	varchar(100)			Content - Reviews
element	varchar(100)			reviews
folder	varchar(100)			content
access	tinyint(3) unsigned			0
ordering	int(11)			0
published	tinyint(3)			1
iscore	tinyint(3)			0
client_id	tinyint(3)			0
checked_out	int(11) unsigned			0
checked_out_time	datetime			0000-00-00 00:00:00
params	text			

Using phpMyAdmin, pull up a screen to enter rows into `jos_plugins` and enter the information as in the following screenshot to register the **Content – Review Information** plug-in in the database:



Field	Type	Function	Null	Value
id	int(11)			
name	varchar(100)			Content - Review Information
element	varchar(100)			reviewinfo
folder	varchar(100)			content
access	tinyint(3) unsigned			0
ordering	int(11)			0
published	tinyint(3)			0
iscore	tinyint(3)			0
client_id	tinyint(3)			0
checked_out	int(11) unsigned			0
checked_out_time	datetime			0000-00-00 00:00:00
params	text			

Using phpMyAdmin, pull up a screen to enter rows into `jos_plugins` and enter the information as in the following screenshot to register the **Search – Reviews** plug-in in the database:



Field	Type	Function	Null	Value
id	int(11)			
name	varchar(100)			Search - Reviews
element	varchar(100)			reviews
folder	varchar(100)			search
access	tinyint(3) unsigned			0
ordering	int(11)			0
published	tinyint(3)			1
iscore	tinyint(3)			0
client_id	tinyint(3)			0
checked_out	int(11) unsigned			0
checked_out_time	datetime			0000-00-00 00:00:00
params	text			

A Simple Link Plug-In

One of our critics suggested that we should code something that would allow them to link to a review by simply typing in the restaurant name. For instance, while writing an article about lunch spots, the critic doesn't want to hunt down a link to the *Crosstown Deli* review. Instead, by just typing *Crosstown Deli,* he wants it to turn into a link when the article is published. We could modify the code in `com_content` to do this, but the other critics don't know if they want to commit to this system yet. Also, if `com_content` is ever patched, we'll have to modify the code again. Instead, we will create a plug-in to search the output for review titles and automatically turn them into links. To do this, create the `reviews.php` file in the `/plugins/content` in your Joomla! installation and add the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
$mainframe->registerEvent( 'onPrepareContent',
                                'pluginReviews' );
```

We use the `registerEvent()` function of the `$mainframe` object to assign `pluginReviews()` to the `onPrepareContent` event. When Joomla! loads a content item from the database, it will trigger all the functions assigned to the `onPrepareContent` event.

The `pluginReviews()` function is passed the row for the article being loaded, along with an object of parameters for the article. We use `contentReviews_getlist()` to get an array of restaurant names keyed by review `id`. Next, we build two arrays: one with patterns to search for and another with replacement strings. In the `$pattern` array, we're using `preg_quote()` on all the restaurant names to escape any characters that would normally be a part of a regular expression. For `$replace`, we call `contentReviews_makeLink()` and pass in the review name as well as the array of names. This will allow `contentReviews_makeLink()` to extract the `id` and format the link. Once our patterns and replacements are set, we use `preg_replace()` to substitute the links. We set `$row->text` to the result of `preg_replace()` as the object is passed by reference.

```
function contentReviews_makeLink ($title, &$reviews)
{
  $id = array_search($title, $reviews);
  $link = JRoute::_('index.php?option=com_reviews&view=review&id=' .
$id );
  $link = '<a href="' . $link . '">' . $title . '</a>';
  return $link;
}
```

Given the review title and an array of titles keyed by review id, contentReviews_makeLink() uses PHP's array_search() function to get the id that goes with the title. The variable $link is then set with a relative URL to the article passed through JRoute::_(). The $link variable is then set again with HTML for an anchor tag, using the article title as the link text. We then return $link.

```
function contentReviews_getlist()
{
  $reviews = array();
  $db =& JFactory::getDBO();
  $query = "SELECT id, name FROM #__reviews";
  $db->setQuery($query);
  $rows = $db->loadObjectList('id');
  foreach($rows as $id => $row)
  {
    $reviews[$id] = $row->name;
  }
  return $reviews;
}
?>
```

In contentReviews_getlist(), we pass the column id into loadObjectList() so that the results come back automatically keyed; we only need to reference the name from each row.

Notice that the main function is in the format of 'plug-in' followed by the name of the file. Similarly, the other two functions begin with the type of plug-in (content), followed by the filename, an underscore and a name. While this convention is not required, it will help us avoid name conflicts with other plug-ins.

What Events Can be Registered?

Plug-ins in Joomla! can respond to any number of events during a request. Multiple plug-ins can respond to any one event in their group. The ordering of the plug-ins in Joomla!'s back end determines the order which registered functions are called. For instance, if both plug-in A and plug-in B respond to `onBeforeDisplayContent`, A's function registered with `onBeforeDisplayContent` will be called first. A listing of when these events occur, grouped by plug-in type is as follows:

System:

`onAfterInitialise` – after the framework loads, but before routing and output

`onAfterRoute` – after routing, but before output

`onAfterDispatch` – after the Joomla! application is started

`onAfterRender` – after all output is processed

`onGetWebServices` – when the XML-RPC function requests a list of valid function calls

`onLoginFailure` – when a login attempt fails

Search:

`onSearch` – when a search is performed

`onSearchAreas` – when the search component requests a list of valid search areas

Authentication:

`onAuthenticate` – when a user initially attempts to authenticate, provides a method for authentication

User:

`onLoginUser` – after a user initally authenticates, but before fully logged in: all functions must return true finish to authenticate
`onLogoutUser` – when a user attempts to logout: all functions must return true to logout
`onBeforeStoreUser` – just before a user is stored in the database
`onAfterStoreUser` – after a user is stored in the database
`onBeforeDeleteUser` – just before a user is deleted from the system
`onAfterDeleteUser` – just after a user is deleted from the system

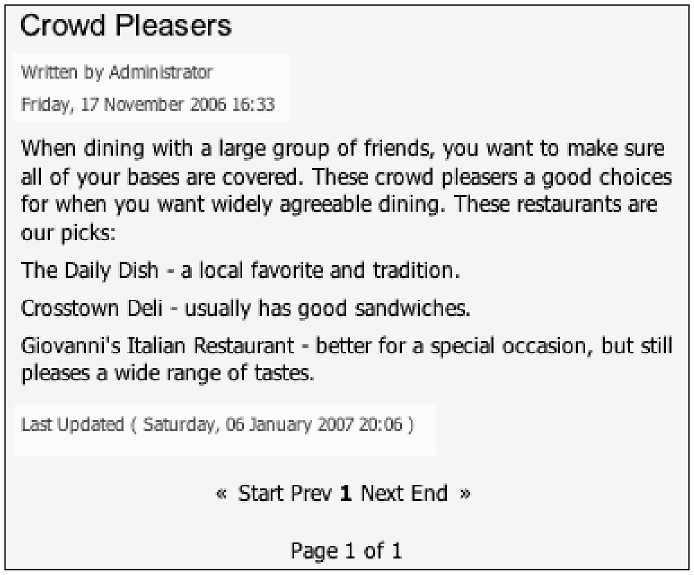
Editor-xtd:
`onCustomEditorButton` – when custom editor buttons are loaded, allows the additon of buttons

Editor:
`onInit` – when the editor is initialized
`onDisplay` – when the editor is ready to be displayed
`onGetContent` – when the contents of the editor are requested
`onSetContent` – when the contents of the editor are populated
`onSave` – when the contents of the editor are saved
`onGetInsertMethod` – just before the editor is output

Content:
`onPrepareContent` – before any output occurs
`onAfterDisplayTitle` – just after article title is displayed
`onBeforeDisplayContent` – just before content is output, returns output to be displayed
`onAfterDisplayContent` – just after content is output, returns output to be displayed

```
function pluginReviews( &$row, &$params )
{
  $reviews = contentReviews_getlist();
  $pattern = array();
  $replace = array();
  foreach($reviews as $review)
  {
    $pattern[] = '/' . preg_quote($review) . '/';
    $replace[] = contentReviews_makeLink($review,
$reviews);
  }
  $row->text = preg_replace($pattern, $replace, $row-
>text);
  return true;
}
```

Before adding the plug-in, one of our articles could have looked like the following:



Crowd Pleasers

Written by Administrator

Friday, 17 November 2006 16:33

When dining with a large group of friends, you want to make sure all of your bases are covered. These crowd pleasers a good choices for when you want widely agreeable dining. These restaurants are our picks:

The Daily Dish - a local favorite and tradition.

Crosstown Deli - usually has good sandwiches.

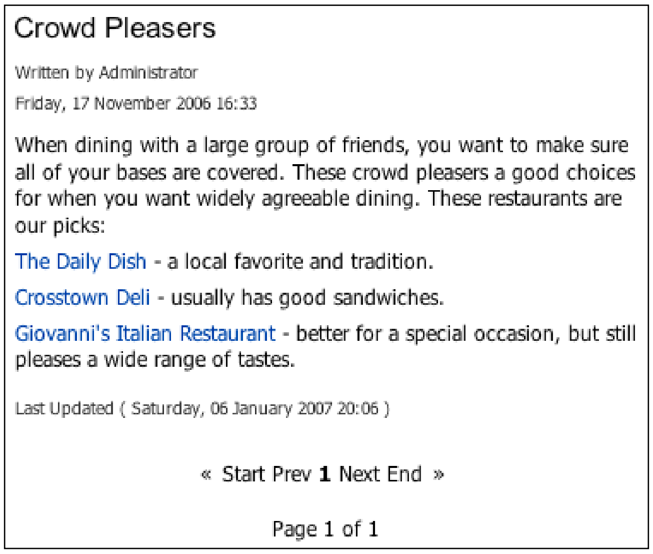
Giovanni's Italian Restaurant - better for a special occasion, but still pleases a wide range of tastes.

Last Updated (Saturday, 06 January 2007 20:06)

« Start Prev **1** Next End »

Page 1 of 1

After applying the plug-in, our article will change the restaurant names into links like in the following screenshot:



Crowd Pleasers

Written by Administrator

Friday, 17 November 2006 16:33

When dining with a large group of friends, you want to make sure all of your bases are covered. These crowd pleasers a good choices for when you want widely agreeable dining. These restaurants are our picks:

The Daily Dish - a local favorite and tradition.

Crosstown Deli - usually has good sandwiches.

Giovanni's Italian Restaurant - better for a special occasion, but still pleases a wide range of tastes.

Last Updated (Saturday, 06 January 2007 20:06)

« Start Prev **1** Next End »

Page 1 of 1

An Information Box Plug-In

Another critic was less interested in the links, but was interested in getting a box that would display the "vital details" for the restaurant of her choice. To use this feature, we will instruct the critic to enclose the name of the review in curly braces, preceded by the word `review` and a space. For example, the details for "The Daily Dish" could be added in by entering `{review The Daily Dish}`.

Why Curly Braces?

Many core plug-ins use curly braces as a way of creating "plug-in tags" in content items so they aren't confused with HTML or XML. Frequently, you will see them used alone (as in `{runmyplugin}`), with parameters (as we're doing with the reviews), or enclosing text (`{plugin}` like this `{/plugin}`). PHP's Perl-style regular expression functions are very useful for detecting these patterns. More information about these functions can be found on the PHP website: `http://www.php.net/manual/en/ref.pcre.php`.

Before we write the code for this, go to the administrator back end, unpublish our first plug-in, and then publish the new one. Go to **Extensions | Plugin Manager**, page through the results, unpublish **Content – Reviews**, and publish **Content – Review Information**.

Then create the file `reviewinfo.php` in `/plugins/content` and add this code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
$mainframe->registerEvent( 'onPrepareContent',
                            'pluginReviewInfo' );
```

As we did for the first plug-in, we register the function `pluginReviewInfo()` to trigger on the `onPrepareContent` event.

```
function pluginReviewInfo ( &$row, &$params )
{
  preg_match_all('/\{reviewinfo (.*)\}/U',
                      $row->text, $matches);
  foreach( $matches[1] as $name )
  {
    $review = contentReviewInfo_getReviewByName($name);
    $html = contentReviewInfo_createHTML($review);
    $row->text = str_replace("{reviewinfo $name}",
                                $html, $row->text);
  }
  return true;
}
```

The triggering event automatically passes the row of content and article parameters into `pluginReviewInfo()`. We use PHP's `preg_match_all()` to get all the `{reviewinfo ...}` tags in the article, collecting them in `$matches`. The array in `$matches[1]` contains all the names that were captured between the space after `reviewinfo` and the end of the tag. We cycle through this array and pass the names into `contentReviewInfo_getReviewByName()` to get the information for each review. Next, we get an HTML-formatted snippet of information in `$html` by passing the `$review` object into `contentReviewInfo_createHTML()`. Finally, we use PHP's `str_replace()` to replace all occurrences of the `reviewinfo` tag for that review with the HTML snippet.

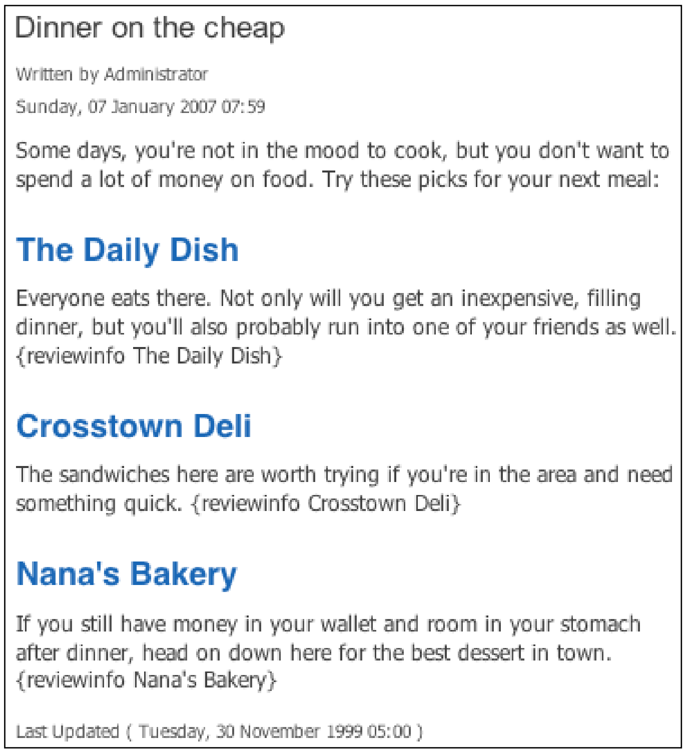
```
function contentReviewInfo_getReviewByName ($name)
{
  $db =& JFactory::getDBO();
  $name = addslashes($name);
  $query = "SELECT * FROM #__reviews WHERE name = '$name'";
  $db->setQuery($query);
  $review = $db->loadObject();
  return $review;
}
```

Without a review `id`, we use `contentReviewInfo_getReviewByName()` to fetch the information for the review given only the name. First, we get a reference to the current database object. Next, we take the `$name` variable pass it into the function and run it through PHP's `addslashes()` so that reviews with apostrophes do not cause the query to fail. We run the query and use the member function `loadObject()` to load only the first row in the results. We've warned our critic that this may not work as expected if two people write separate reviews for the same restaurant, but she's guaranteed us that the other critics are too narcissistic to write about a place someone else already reviewed.

```
function contentReviewInfo_createHTML (&$review)
{
  $html = '<table class="moduletable">';
  $html .= '<tr><th colspan="2">Info</th></tr>';
  $html .= '<tr><td>Address:</td><td>' .
                $review->address . '</td></tr>';
  $html .= '<tr><td>Price Range:</td><td>$' .
       $review->avg_dinner_price . '</td></tr>';
  $html .= '<tr><td>Reservations:</td><td>' .
           $review->reservations . '</td></tr>';
  if ( $review->smoking == 0 )
  {
    $smoking = 'No';
  }
  else
  {
    $smoking = 'Yes';
  }
  $html .= '<tr><td>Smoking:</td><td>' .
                        $smoking . '</td></tr>';
  $html .= '</table>';
  return $html;
}
?>
```

Finally, our `contentReviewInfo_createHTML()` function takes a review object row as a parameter and formats it into an HTML table. This HTML table is given the `moduletable` class, which is standard in Joomla! templates. The `address`, `price range`, and `reservations` policy are all included in the HTML as-is. The `smoking` column is tested, with `$smoking` set to `Yes` or `No` depending on the column value. We then use `$smoking` to finish the last row in the table.

Before applying this plug-in, our critic's article would probably look something like the following screenshot:



Dinner on the cheap

Written by Administrator

Sunday, 07 January 2007 07:59

Some days, you're not in the mood to cook, but you don't want to spend a lot of money on food. Try these picks for your next meal:

The Daily Dish

Everyone eats there. Not only will you get an inexpensive, filling dinner, but you'll also probably run into one of your friends as well. {reviewinfo The Daily Dish}

Crosstown Deli

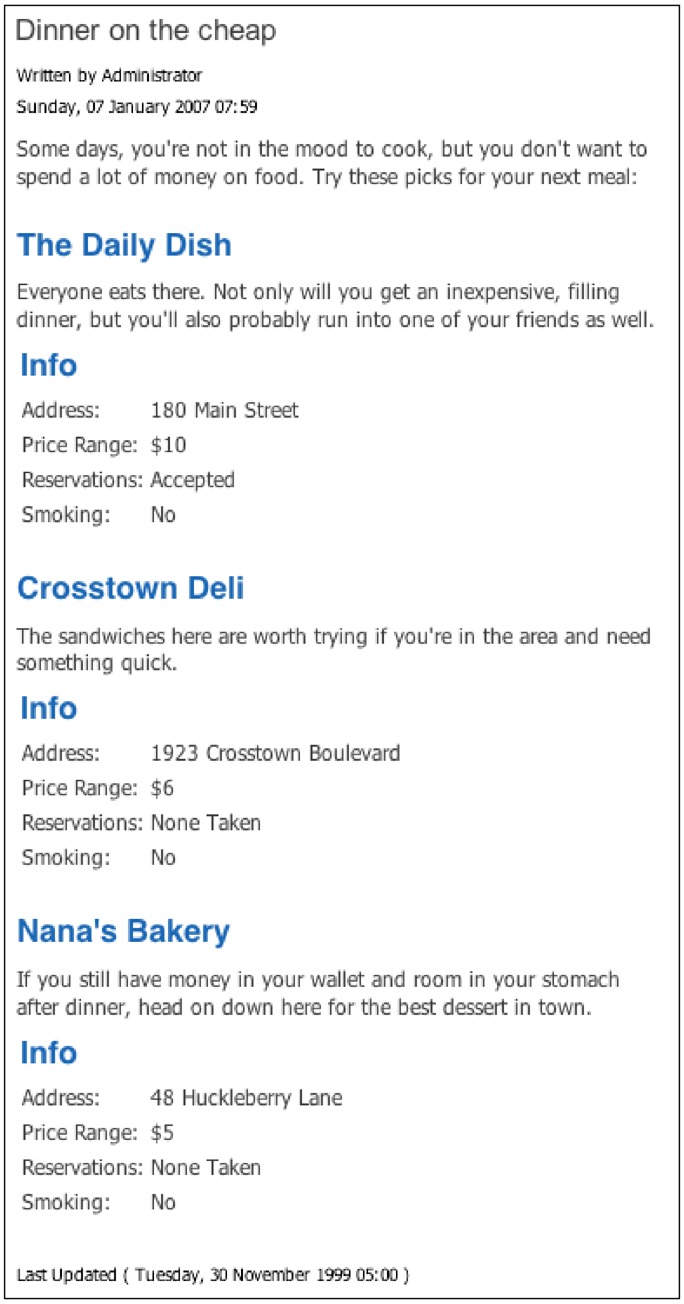
The sandwiches here are worth trying if you're in the area and need something quick. {reviewinfo Crosstown Deli}

Nana's Bakery

If you still have money in your wallet and room in your stomach after dinner, head on down here for the best dessert in town. {reviewinfo Nana's Bakery}

Last Updated (Tuesday, 30 November 1999 05:00)

After publishing the plug-in, the article will transform into something like the following:



Dinner on the cheap

Written by Administrator

Sunday, 07 January 2007 07:59

Some days, you're not in the mood to cook, but you don't want to spend a lot of money on food. Try these picks for your next meal:

The Daily Dish

Everyone eats there. Not only will you get an inexpensive, filling dinner, but you'll also probably run into one of your friends as well.

Info

Address:	180 Main Street
Price Range:	$10
Reservations:	Accepted
Smoking:	No

Crosstown Deli

The sandwiches here are worth trying if you're in the area and need something quick.

Info

Address:	1923 Crosstown Boulevard
Price Range:	$6
Reservations:	None Taken
Smoking:	No

Nana's Bakery

If you still have money in your wallet and room in your stomach after dinner, head on down here for the best dessert in town.

Info

Address:	48 Huckleberry Lane
Price Range:	$5
Reservations:	None Taken
Smoking:	No

Last Updated (Tuesday, 30 November 1999 05:00)

Searching the Reviews

Some of our visitors have complained that they cannot find restaurant reviews through the search box at the top of the screen. We can fix this by writing a search plug-in to scan the reviews along with the results for content. Create the file `/plugins/search/reviews.php` and add the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
$mainframe->registerEvent( 'onSearch', 'botSearchReviews' );
$mainframe->registerEvent( 'onSearchAreas', 'botSearchReviewAreas' );
```

For the search plug-in, we've registered functions for two events: `onSearch` and `onSearchAreas`. The `onSearch` event is triggered when the search component looks for results for a given phrase. Triggers to the `onSearchAreas` event are made when the search form is being built so that a visitor can have a choice over which records are returned.

```
function &botSearchReviewAreas() {
  static $areas = array(
    'reviews' => 'Restaurant Reviews'
  );
  return $areas;
}
```

The `botSearchReviewAreas()` function is used to return an array of valid search areas. While it is possible to have more than one search area (say, search the review text or search the review notes), we are going to keep it simple for visitors and just search reviews as a whole. By default, the search component will return results from all published search plug-ins. When the visitor checks off one or more search areas, they are limiting their search to only find records from those areas. An ampersand precedes the function name `botSearchReviewAreas()` so that a reference to the `$areas` array is returned; we don't want to create a redundant copy of the static array in memory.

```
function botSearchReviews ( $text, $phrase='',
                            $ordering='', $areas=null )
{
```

When a search takes place, `botSearchReviews()` is called with up to four parameters. The search keywords are passed into `$text`. We can impose special conditions using `$phrase`, which can be set to `any` (any of these words), `all` (all of these words), or `exact` (exactly this phrase). Record sorting is determined by `$ordering`, which can be set to `newest`, `oldest`, `popular`, `alpha`, or `category`. Finally, `$areas` is an array of the search areas currently selected by the visitor.

```
    if (!$text) {
      return array();
    }
    if (is_array( $areas )) {
      if (!array_intersect( $areas,
                 array_keys( botSearchReviewAreas() ) )) {
        return array();
      }
    }
```

There are a couple of situations where we may want to quit the processing at that instant. First, if there are no search keywords in `$text`, we know there won't be any results, so we return an empty array. Next, we test to see if `$areas` is set as an array. If so, we match this array against the valid search areas for our plug-in. If none of the items in `$areas` are intended for our plug-in, we return an empty array. When `$areas` is set to `null`, all search areas are assumed to be on.

```
    $db =& JFactory::getDBO();
    if ($phrase == 'exact')
    {
      $where = "(LOWER(name) LIKE '%$text%')
            OR (LOWER(quicktake) LIKE '%$text%')" .
          " OR (LOWER(review) LIKE '%$text%')
            OR (LOWER(notes) LIKE '%$text%')";
    }
    else
    {
      $words = explode( ' ', $text );
      $wheres = array();
      foreach ($words as $word) {
        $wheres[] = "(LOWER(name) LIKE '%$word%')
                OR (LOWER(quicktake) LIKE '%$word%')" .
              " OR (LOWER(review) LIKE '%$word%')
                OR (LOWER(notes) LIKE '%$word%')";
      }
      if($phrase == 'all')
      {
        $separator = "AND";
      }
      else
      {
        $separator = "OR";
      }
      $where = '(' . implode( ") $separator (" , $wheres ) . ')';
    }
    $where .= ' AND published = 1';
```

After getting the current database object instance from `JFactory::getDBO()`, we build the `WHERE` clause for a query to use for searching the reviews. We test the variable `$phrase` to determine how the visitor wants to have the search terms treated. If they were intending an exact phrase, we simply match the search term as a whole against the `name`, `quicktake`, `review`, and `notes` fields in `jos_reviews`. For 'all' and 'any' searches, we separate out each word in the search term and build an array of `WHERE` statements, which are joined according to the type of search. Finally, we add a check to make sure we only include published reviews.

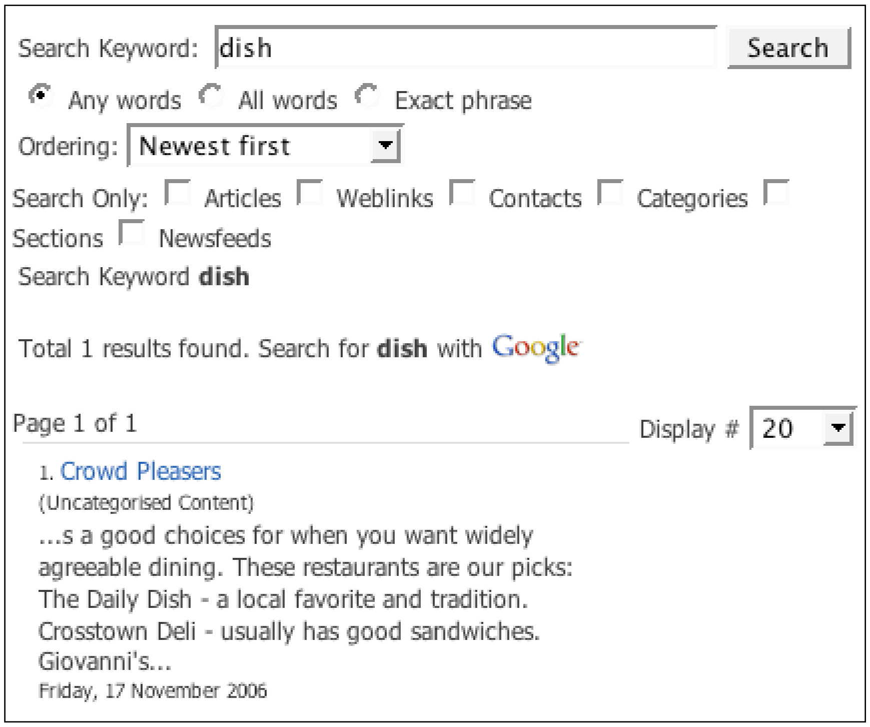
```
    switch ($ordering) {
      case 'oldest':
        $order = 'review_date ASC';
        break;
      case 'alpha':
        $order = 'title ASC';
        break;
      case 'newest':
      default:
        $order = 'review_date DESC';
        break;
    }
```

After dealing with the search terms, we need to order the reviews properly. Of the five possible, only three really make sense for our plug-in: oldest, newest, and alphabetical. Our default is to sort by the review date reverse chronologically.

```
    $query = "SELECT name AS title, quicktake AS text,
                           review_date AS created, " .
    "\n 'Restaurant Reviews' AS section," .
    "\n CONCAT('index.php?option=com_reviews&view=review&id=', id) AS
  href," .
    "\n '2' AS browsernav" .
    "\n FROM #__reviews" .
    "\n WHERE $where" .
    "\n ORDER BY $order";
    $db->setQuery( $query );
    $rows = $db->loadObjectList();
    return $rows;
  }
  ?>
```

Finally, we finish assembling our query. Links to reviews are constructed directly in the query using the `CONCAT()` function and are aliased to the `href` column. We also alias the value `2` as `browsernav`. Also, **name** is aliased to **title**, **quicktake** to **text** and **review_date** to **created**. All these column aliases are expected by the search component so that it knows which links to output and how to format them. When `browsernav` is set to `1`, links are set to open in a new window; when it is set to `2`, they open in the current one.

The query is then set, we load the results as an array of objects into `$rows`, and we return `$rows` so that they can be included alongside other search results. Before the addition of this plug-in, a search on 'dish' might have returned a result set similar to the following one:



Search Keyword: dish

[✓ Any words] All words Exact phrase

Ordering: Newest first

Search Only: Articles Weblinks Contacts Categories Sections Newsfeeds

Search Keyword **dish**

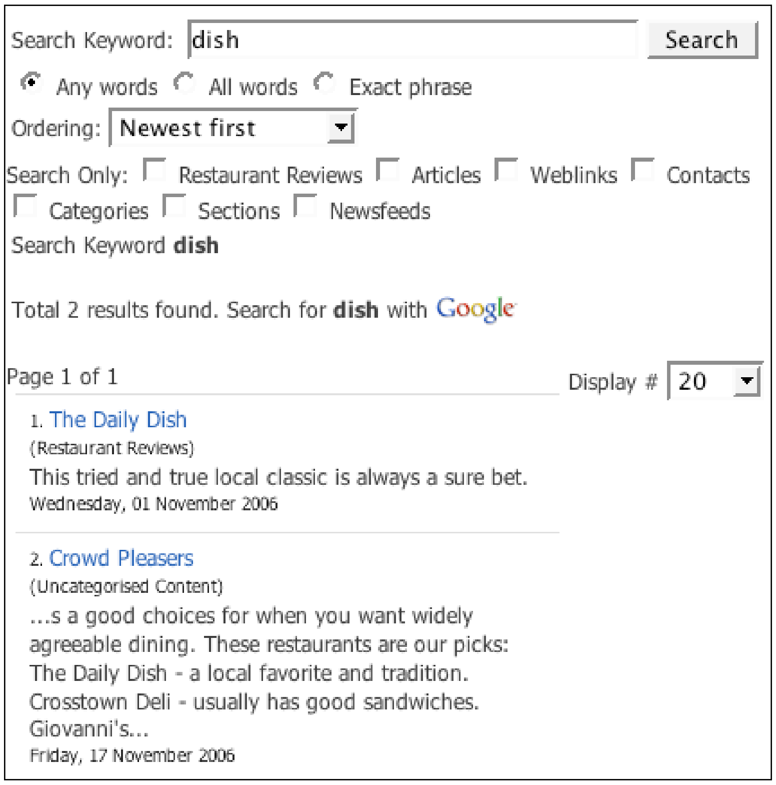
Total 1 results found. Search for **dish** with Google

Page 1 of 1

Display # 20

1. Crowd Pleasers
(Uncategorised Content)
...s a good choices for when you want widely agreeable dining. These restaurants are our picks: The Daily Dish - a local favorite and tradition. Crosstown Deli - usually has good sandwiches. Giovanni's...
Friday, 17 November 2006

After adding the plug-in, "The Daily Dish" should appear in the results:



Search Keyword: dish [Search]

(● Any words) ○ All words ○ Exact phrase

Ordering: Newest first

Search Only: ☐ Restaurant Reviews ☐ Articles ☐ Weblinks ☐ Contacts ☐ Categories ☐ Sections ☐ Newsfeeds

Search Keyword **dish**

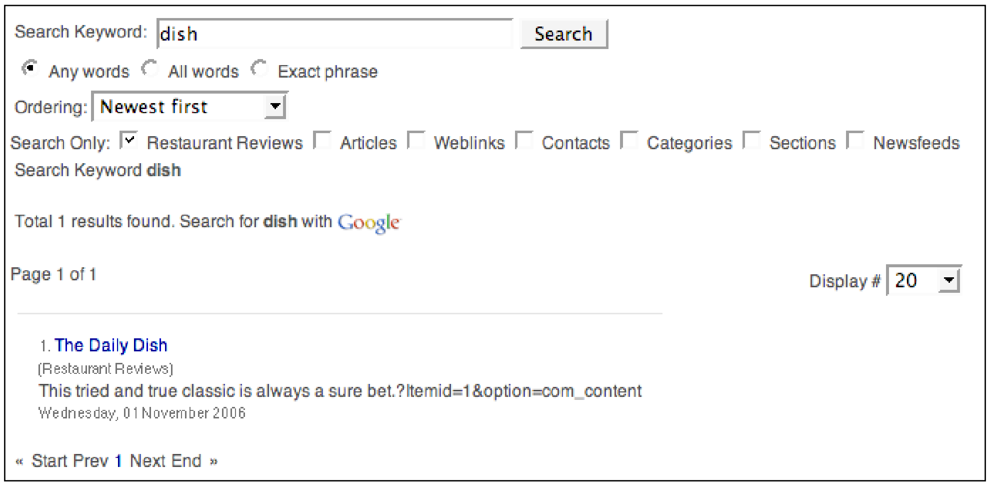
Total 2 results found. Search for **dish** with Google

Page 1 of 1 Display # 20

1. The Daily Dish
(Restaurant Reviews)
This tried and true local classic is always a sure bet.
Wednesday, 01 November 2006

2. Crowd Pleasers
(Uncategorised Content)
...s a good choices for when you want widely agreeable dining. These restaurants are our picks: The Daily Dish - a local favorite and tradition. Crosstown Deli - usually has good sandwiches. Giovanni's...
Friday, 17 November 2006

Limiting the search to only reviews should return just "The Daily Dish":



Search Keyword: dish [Search]

(● Any words) ○ All words ○ Exact phrase

Ordering: Newest first

Search Only: [✓ Restaurant Reviews] ☐ Articles ☐ Weblinks ☐ Contacts ☐ Categories ☐ Sections ☐ Newsfeeds

Search Keyword **dish**

Total 1 results found. Search for **dish** with Google

Page 1 of 1 Display # 20

1. The Daily Dish
(Restaurant Reviews)
This tried and true classic is always a sure bet.?Itemid=1&option=com_content
Wednesday, 01 November 2006

« Start Prev 1 Next End »

Summary

With our plug-ins in place, the critics are now able to link to the original reviews without even copying URLs. We've also given them the option of showing a details box that matches whichever template we choose. Finally, the reviews can now be found in search results along with our articles. Visitors can limit the keyword search to just the reviews if they wish; if they remember the word 'croutons' being in a certain review, they will have no problems finding it.

8
Configuration Settings

Our reviewers are satisfied with all the features we've provided, but there are some concerns. They would like to have specific control over certain functions. Fortunately, we don't have to add any mundane record management to do this; instead we can concentrate on the logic. Our extensions will need a little rewriting, but nothing too drastic.

- Adding Parameters to Extensions
- Parameters for Modules
- Parameters for Plug-ins
- Parameters for Components

Adding Parameters to Extensions

Throughout the book, we've run queries to register extensions in Joomla!. Within the tables where we inserted this data, there is a column named `params`. This column allows us to store configuration parameter values in the database. However, the column itself does not enforce any kind of format for the parameter values.
To do this, we will enter the parameter values as a part of an XML document for configuration. This file will sit alongside our main module file and contain basic identification information, along with a list of all the possible settings.

Parameters for Modules

The module we wrote for Restaurant Reviews already has logic for different display types and different data retrieval scenarios. This will make it simpler to pull in the parameters and use them in a meaningful way.

In /modules/mod_reviews create an XML configuration file mod_reviews.xml and enter the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<install type="module" version="1.5">
  <name>Restaurant Reviews</name>
  <author>Sumptuous Software</author>
  <creationDate>January 2007</creationDate>
  <copyright>(C) 2007</copyright>
  <license>Commercial</license>
  <authorEmail>support@packtpub.com</authorEmail>
  <authorUrl>www.packtpub.com</authorUrl>
  <version>1.0</version>
  <description>A module for promoting restaurant
                    reviews.</description>
  <params>
    <param name="random" type="radio" default="0" label="Randomize"
                    description="Show random reviews">
      <option value="0">No</option>
      <option value="1">Yes</option>
    </param>
    <param name="@spacer" type="spacer"
                    default="" label="" description="" />
    <param name="items" type="text" default="1" label="Display #"
                    description="Number of reviews to display" />
    <param name="style" type="list" default="default" label="Display
style" description="The style to use for displaying the reviews.">
      <option value="default">Flat</option>
      <option value="bulleted">Bulleted</option>
    </param>
  </params>
</install>
```

The XML document begins with a standard XML definition, with all the remaining elements wrapped by `<install>`. This first element defines that the extension we're describing is a module and that it is intended for Joomla!. Within `<install>`, we have several elements that are intended for identification: `name`, `author`, `creation date`, `copyright`, `license`, `author email`, `author URL`, `version`, and `description`. Except for `description`, these will appear in the back end in the modules section of **Extension Manager**.

After the identification elements, we then add the `<params>` element, which encases several `<param>` elements. For our module we would like to provide options for controlling display of random restaurants, number of reviews displayed at a time, and whether the review display is flat listed or bulleted.

The XML document not only allows us to enforce the data rules for these options, but also allows us to define how they would appear as back-end controls. The control for randomizing the reviews makes sense as a yes/no decision, so a radio button is appropriate. We give these parameters the name `random` (to match our earlier code), the type `radio`, the default `0` (for no), the label `Randomize`, and the description `Show random reviews`. Within this parameter element, we define two `<option>` elements, one with `0` as a value and `No` as the text and another with `1` and `Yes` respectively, similar to the way in which HTML select options are coded.

Since the other two options have less to do with data retrieval and more with display, we will set these options off with a spacer. The spacer will not have a description, label, or default value, but will be of type `spacer` and named `@spacer`.

For the number of items to be displayed, we want the administrator to simply enter in a number. We use a parameter of type `text` and set the default to `1`. For the choice of style, there should only be a choice from the ones available. We use the list type of parameter and define the `flat` and `bulleted` options similarly to the way we did for the radio button.

What Parameters Are Available for Use?

Modules, plug-ins, and components all allow you to define configuration parameters through XML files. Many common parameter types are predefined and can be used in any extension. Every parameter you define must have five basic attributes. First, you must give the parameter a name so you can reference it later in your code. Next, you need a default value to be displayed and used if no value is chosen. To identify the parameter, you need to give it both a visible label and a description appearing when the mouse cursor hovers over it. Finally, you must specify the type of parameter; a categorized list is below:

Content:

`section` - All published sections in a list

`category` - All published categories in a list

Text Input:

`text` - A standard text input

`textarea` - A plain textarea field

`password` - A standard text input where the characters are masked as they are entered

`editors` - Provides the admin's currently chosen WYSIWYG editor for input

Selections:

menu - All published menus in a list

menuitem - All published menu items in a list `filelist` - A list of files to choose from, given a base folder path

`folderlist` - A list of folders to choose from, given a base folder path

`imagelist` - A list of images to choose from, given a base folder path

`list` - A list of items to choose from (hardcoded into parameter definition)

`radio` - A list of radio selection items to choose from (hardcoded into parameter definition)

`sql` - Creates a dropdown list out of a provided SQL query

Predefined:

`helpsites` - A list of websites powering help file translations to choose from

languages - A list of installed languages to choose from

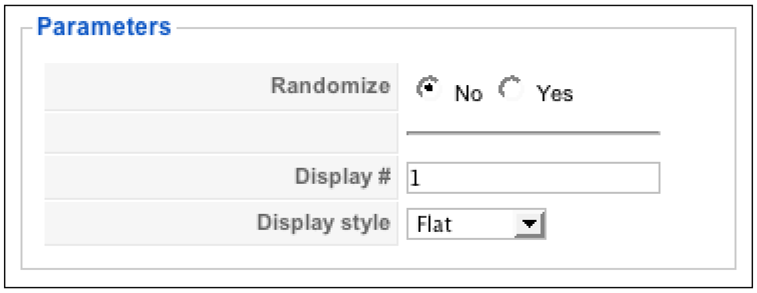
spacer - Creates a visual separation between parameters; no input value is required

timezones - A list of all world timezones

Other:

hidden - Creates a hidden form element with the value and name provided

After saving the XML document, go to the back end and navigate to **Extensions | Module Manager**. From here, choose **Restaurant Reviews** from the list and you should see a screen similar to the following:



Parameters

Randomize	[✓ No] Yes
Display #	1
Display style	Flat

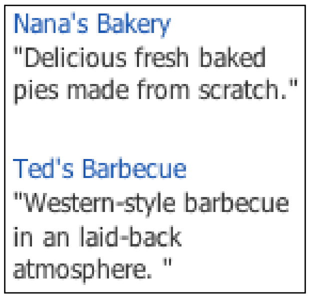
You should be able to set these parameters, save the module, then reopen it and see the changes. With the parameter definitions in place, we now need to make some adjustments to the code so that the values are used in a meaningful way. Open `/modules/mod_reviews/mod_reviews.php` and make the following highlighted changes:

```
<?php
defined('_JEXEC') or die('Restricted access');
require(dirname(__FILE__).DS.'helper.php');
$random = $params->get('random', 0);
$style = $params->get('style', 'default');
```

```
if($random)
{
  $list = modReviewsHelper::getRandomReview();
}
else
{
  $list = modReviewsHelper::getReviews($params);
}
require(JModuleHelper::getLayoutPath('mod_reviews', $style));
?>
```

For modules, `$params` is automatically available in global scope. This object has the member function `get()`, which returns the parameter value given the name (and optionally, a default value). Our previous code handles the `random` option, while we make a small change in the call to `getLayoutPath` to allow for the different styles.

After saving the file, go back to the module configuration panel, **Restaurant Reviews** in the back end, set **Display #** to a value of **2**, then save the module. On the front end, the module should appear similar to the following image:


Nana's Bakery
"Delicious fresh baked pies made from scratch."

Ted's Barbecue
"Western-style barbecue in an laid-back atmosphere. "

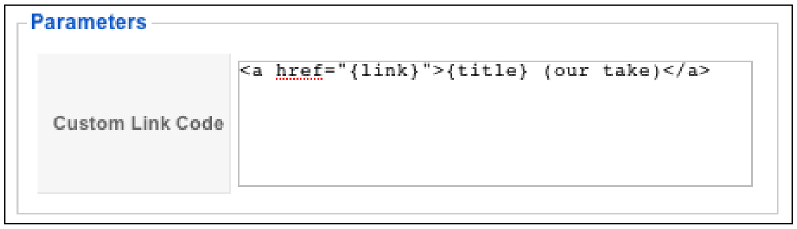
Parameters for Plug-Ins

For our content review links plug-in, we would like to give the administrators some control for formatting the link. They should be able to add text to the link to show that it goes to a review or change the anchor tag to have more attributes. The process for adding parameters to our plug-in is similar to what we do for modules. Open `/plugins/content/reviews.xml` and add this code:

```
<?xml version="1.0" encoding="utf-8"?>
<install version="1.5" type="plugin" group="content">
  <name>Content - Restaurant Review Links</name>
  <author>Sumptuous Software</author>
  <creationDate>January 2007</creationDate>
  <copyright>(C) 2007</copyright>
  <license>Commercial</license>
```

```
    <authorEmail>support@packtpub.com</authorEmail>
    <authorUrl>www.packtpub.com</authorUrl>
    <version>1.0</version>
    <description>Searches for titles of restaurants in
               articles and turns them into review links.</description>
    <params>
      <param name="linkcode" type="textarea" default=""
               rows="5" cols="40" label="Custom Link Code"
               description="By using {link} and {title},
               you can generate custom HTML output that
               includes the URL and review title respectively." />
    </params>
  </install>
```

Since our administrators will be adding a bit of code, it will be helpful to have more room than a typical text input provides. To handle this, we create the `linkcode` parameter as type `textarea`. Go to the back end and navigate to **Extensions | Plugin Manager**, then select **Content – Reviews** from the list. There should be a box labeled **Custom Link Code,** where you can enter the code for the links. Our reviewers have decided that they want to reinforce the fact that the reviews are merely their opinions of the restaurants. They want the text **(our take)** to follow the title of each review. Since they're prone to change their minds, the parameter we're defining will give them a way of changing the output without actually getting into the code. We will define the tags `{link}` and `{title}` as the relative URL and review title respectively; these tags will be dynamically replaced with the appropriate values when the content is displayed. In the **Custom Link Code** box, enter `<a href="{link}">{title} (our take)</a>`. It should look like the following figure:



```
<a href="{link}">{title} (our take)</a>
```

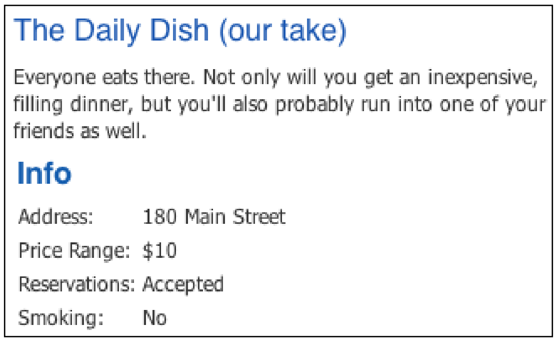
Save the plug-in. Before our formatting can take effect, we need to change the plug-in code to read the parameters and change the link accordingly. Open `/plugins/content/reviews.php` and make the following highlighted changes and additions:

```
function pluginReviews( &$row, &$params )
{
  $plugin =& JPluginHelper::getPlugin('content', 'reviews');
  $pluginParams = new JParameter( $plugin->params );
  $reviews = contentReviews_getlist();
```

```
    $pattern = array();
    $replace = array();
    foreach($reviews as $review)
    {
      $pattern[] = '/' . preg_quote($review) . '/';
      $replace[] = contentReviews_makeLink(
                                $review, $reviews, $pluginParams);
    }
    $row->text = preg_replace($pattern, $replace, $row->text);
    return true;
  }
  function contentReviews_makeLink ($title, &$reviews, &$pluginParams)
  {
    $linkcode = $pluginParams->get('linkcode', '');
    $id = array_search($title, $reviews);
    $link = JRoute::_('index.php?option=com_reviews&view=review&id=' .
  $id);
    if($linkcode == '')
    {
      $linkcode = '<a href="' . $link . '">' . $title . '</a>';
    }
    else
    {
      $linkcode = str_replace('{link}', $link, $linkcode);
      $linkcode = str_replace('{title}', $title, $linkcode);
    }
    return $linkcode;
  }
```

In `pluginReviews()`, we start by using the `getPlugin()` member function of `JPluginHelper` to get our plug-in object, passing in the plug-in folder `content` and name `reviews` respectively. The `params` member variable is passed into a constructor for `JParameter` to get the parameters back as an object. We pass this object into `contentReviews_makeLink()`, where the value for the `linkcode` parameter is extracted. If there is no value for `linkcode`, we return the link as usual. Otherwise, we look for our `{link}` and `{title}` tags and replace them with the appropriate elements.

After saving the plug-in code, links to reviews should look similar to the following title:



The Daily Dish (our take)

Everyone eats there. Not only will you get an inexpensive, filling dinner, but you'll also probably run into one of your friends as well.

Info

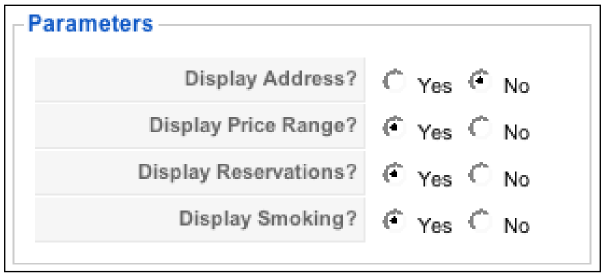
Address:	180 Main Street
Price Range:	$10
Reservations:	Accepted
Smoking:	No

Adding configuration for the `Review Information` plug-in is similar. This time, we have four possible pieces of data that are displayed with every information box. This might be too much, so we will allow the administrator to turn certain fields off. Open `/plugins/content/reviewinfo.xml` and add the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<install version="1.5" type="plugin" group="content">
  <name>Content - Review Information</name>
  <author>Sumptuous Software</author>
  <creationDate>January 2007</creationDate>
  <copyright>(C) 2007</copyright>
  <license>Commercial</license>
  <authorEmail>support@packtpub.com</authorEmail>
  <authorUrl>www.packtpub.com</authorUrl>
  <version>1.0</version>
  <description>Turns {reviewinfo Name of your restaurant}
              into a table with the review's essential
              details.</description>
  <params>
    <param name="address" type="radio" default="1"
              label="Display Address?" description="Toggles
              the display of the address in summaries.">
      <option value="1">Yes</option>
      <option value="0">No</option>
    </param>
    <param name="price_range" type="radio" default="1"
              label="Display Price Range?" description="Toggles
              the display of the price range in summaries.">
      <option value="1">Yes</option>
      <option value="0">No</option>
    </param>
```

```
        <param name="reservations" type="radio" default="1"
                  label="Display Reservations?" description="Toggles
                  the display of reservation policy in summaries.">
          <option value="1">Yes</option>
          <option value="0">No</option>
        </param>
        <param name="smoking" type="radio" default="1"
                  label="Display Smoking?" description="Toggles
                  the display of smoking policy in summaries.">
          <option value="1">Yes</option>
          <option value="0">No</option>
        </param>
      </params>
    </install>
```

The `<params>` section of this XML file defines four radio buttons, all set to `Yes` by default. These will be used to turn the displays of the `address`, `price range`, `reservations`, and `smoking` in the review summary on and off. After saving the file, open the plug-in by navigating to **Extensions | Plugin Management**, then select **Content – Review Information**. After Selecting **No** for **Display Address**, the parameters box should look similar to the following:



Parameters

Display Address?	Yes [✓ No]
Display Price Range?	[✓ Yes] No
Display Reservations?	[✓ Yes] No
Display Smoking?	[✓ Yes] No

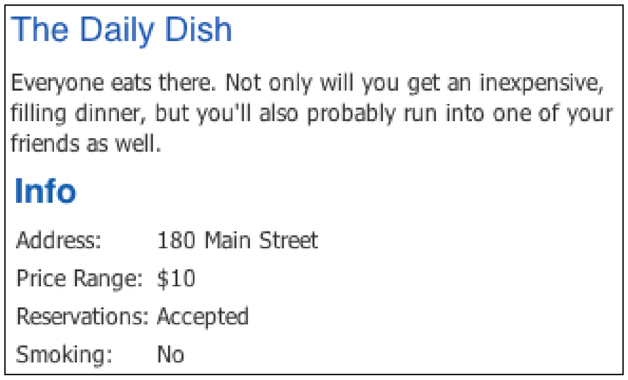
As we did for the link plug-in, we will pull in the parameters once and then pass them into another function. Make the following edits and additions to `/plugins/content/reviewinfo.php`:

```
    function pluginReviewInfo ( &$row, &$params )
    {
      $plugin =& JPluginHelper::getPlugin('content', 'reviewinfo');
      $pluginParams = new JParameter( $plugin->params );
      preg_match_all('/\{reviewinfo (.*)\}/U', $row->text, $matches);
      foreach( $matches[1] as $name )
      {
        $review = contentReviewInfo_getReviewByName($name);
        $html = contentReviewInfo_createHTML($review, $pluginParams);
        $row->text = str_replace("{reviewinfo $name}", $html,
                                 $row->text);
      }
      return true;
    }
    function contentReviewInfo_createHTML (&$review, &$pluginParams)
    {
      $html = '<table class="moduletable">';
      $html .= '<tr><th colspan="2">Info</th></tr>';
      if($pluginParams->get('address', 1))
      {
        $html .= '<tr><td>Address:</td><td>' .
                               $review->address . '</td></tr>';
      }
      if($pluginParams->get('price_range', 1))
      {
        $html .= '<tr><td>Price Range:</td><td>$' .
                               $review->avg_dinner_price . '</td></tr>';
      }
      if($pluginParams->get('reservations', 1))
      {
        $html .= '<tr><td>Reservations:</td><td>' .
                               $review->reservations . '</td></tr>';
      }
      if ( $review->smoking == 0 )
      {
        $smoking = 'No';
      }
      else
      {
        $smoking = 'Yes';
      }
      if($pluginParams->get('smoking', 1))
      {
        $html .= '<tr><td>Smoking:</td><td>' .
                              $smoking . '</td></tr>';
      }
      $html .= '</table>';
      return $html;
    }
```

What about the $params Passed into pluginReviewInfo()?

In the function definition of `pluginReviewInfo()`, a variable named `$params` is passed in. These are not the plug-in's parameters; they are the parameters for the content item. Likewise, the `$row` object is the row in the database in `#__content` matching the current content item.

After getting the parameters for the plug-in in `pluginReviewInfo()`, we pass them into `contentReviewInfo_createHTML()` where we test each field for the corresponding configuration value. If all the fields are set to `Yes`, the information box in content should appear like the following image:



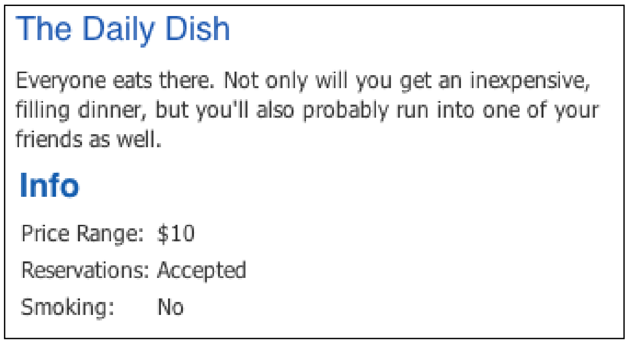
The Daily Dish

Everyone eats there. Not only will you get an inexpensive, filling dinner, but you'll also probably run into one of your friends as well.

Info

Address: 180 Main Street
Price Range: $10
Reservations: Accepted
Smoking: No

When we turn off `Addresses`, it should appear like the following image:



The Daily Dish

Everyone eats there. Not only will you get an inexpensive, filling dinner, but you'll also probably run into one of your friends as well.

Info

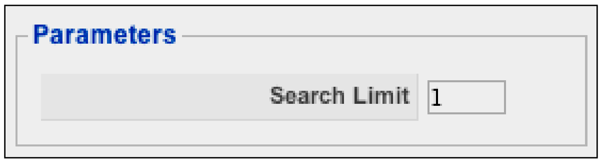
Price Range: $10
Reservations: Accepted
Smoking: No

Finally, we have the search plug-in. Open `/plugins/search/reviews.xml` and add the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<install version="1.5" type="plugin" group="search">
  <name>Search - Restaurant Reviews</name>
  <author>Sumptuous Software</author>
  <creationDate>January 2007</creationDate>
  <copyright>(C) 2007</copyright>
  <license>Commercial</license>
  <authorEmail>support@packtpub.com</authorEmail>
  <authorUrl>www.packtpub.com</authorUrl>
```

```
    <version>1.0</version>
    <description>Allows Searching of Restaurant Reviews</description>
    <params>
      <param name="search_limit" type="text" size="5" default="50"
            label="Search Limit" description="Number of Search items
            to return"/>
    </params>
  </install>
```

By default, we set the number of items to return in the search to `50` (probably more than the number of total items per screen in the search results). With this configuration parameter, we can limit the number of reviews returned all the way down to `1`.

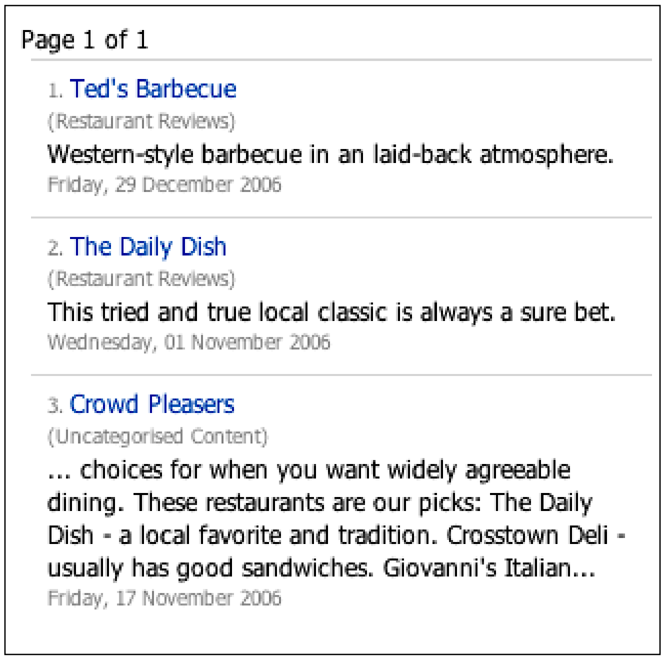


Search Limit: 1

To make our configuration work, open `/plugins/search/reviews.php` and make the following adjustments:

```
function botSearchReviews ( $text, $phrase='',
                    $ordering='', $areas=null )
{
  $db =& JFactory::getDBO();
  if (is_array( $areas ))
  {
    if (!array_intersect( $areas,
                    array_keys( botSearchReviewAreas() ) ))
    {
      return array();
    }
  }
  $plugin =& JPluginHelper::getPlugin('search', 'reviews');
  $pluginParams = new JParameter( $plugin->params );
  $limit = $pluginParams->get( 'search_limit', 50 );
  $db->setQuery( $query, 0, $limit );
  $rows = $db->loadObjectList();
  return $rows;
}
```

As with the other plug-ins, we first get the parameters for search reviews and then use `JParameter` to create an object out of them. For the call to `setQuery()`, we're passing in two additional values that will automatically build our limit clause: `0` to start with the first row and `$limit` to go to our configured limit. Before adding our limit of only one restaurant review per search, a search for 'local' may have resulted in the following screen:



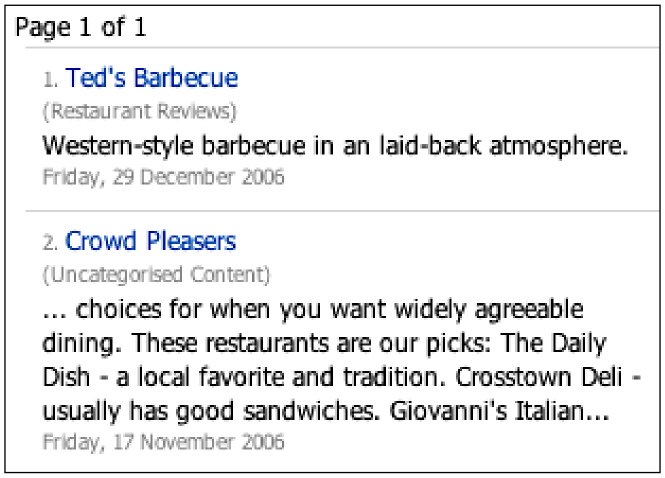
Page 1 of 1

1. Ted's Barbecue
(Restaurant Reviews)
Western-style barbecue in an laid-back atmosphere.
Friday, 29 December 2006

2. The Daily Dish
(Restaurant Reviews)
This tried and true local classic is always a sure bet.
Wednesday, 01 November 2006

3. Crowd Pleasers
(Uncategorised Content)
... choices for when you want widely agreeable dining. These restaurants are our picks: The Daily Dish - a local favorite and tradition. Crosstown Deli - usually has good sandwiches. Giovanni's Italian...
Friday, 17 November 2006

After adding a limit, the results would look more like the following:



Page 1 of 1

1. Ted's Barbecue
(Restaurant Reviews)
Western-style barbecue in an laid-back atmosphere.
Friday, 29 December 2006

2. Crowd Pleasers
(Uncategorised Content)
... choices for when you want widely agreeable dining. These restaurants are our picks: The Daily Dish - a local favorite and tradition. Crosstown Deli - usually has good sandwiches. Giovanni's Italian...
Friday, 17 November 2006

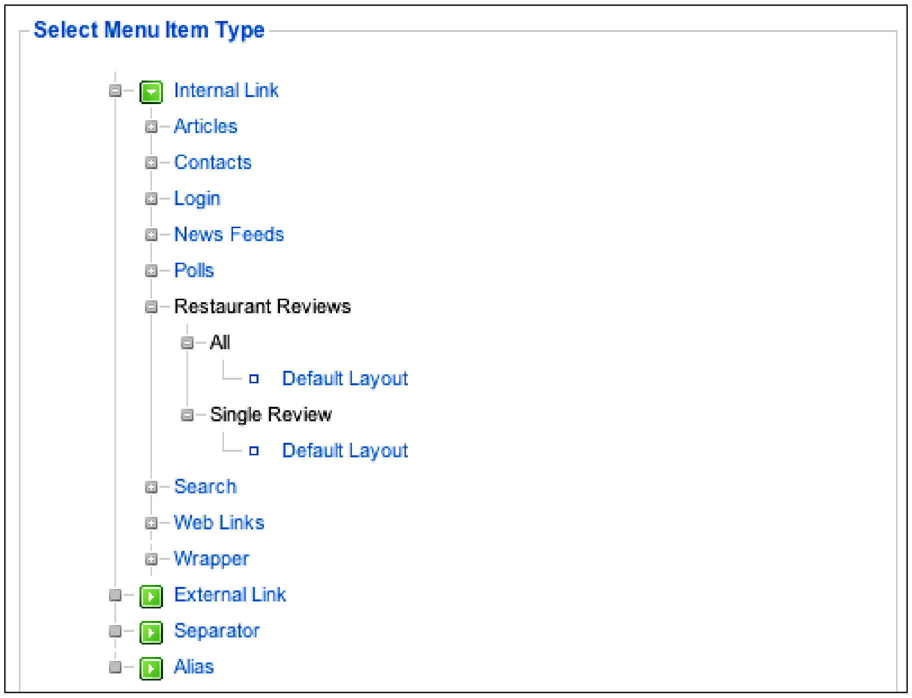
In the two results, the number of content items returned is unchanged; only the number of reviews has changed.

Parameters for Components

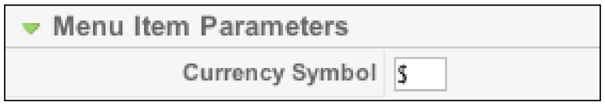
In an effort to internationalize our component, we want to give the administrators the ability to change the currency symbol displayed alongside the `average dinner price` field. Our component is separated into multiple views and we can configure each of them separately. Instead of writing an XML file in the back end, we will make one for each view in the front end. To start, create the `metadata.xml` file in the folder `/components/com_reviews/views/all` and add the following information:

```
<?xml version="1.0" encoding="utf8"?>
<metadata>
  <view title="All">
    <message>
      <![CDATA[Shows all reviews.]]>
    </message>
  </view>
  <params>
    <param name="currency_symbol" type="text" size="3"
                default="$" label="Currency Symbol"
                description="Enter the currency symbol
                to be used for average dinner prices." />
  </params>
</metadata>
```

The XML configuration file for a view is much shorter than the one used for an entire extension. First, we enclose all of our data within the `<metadata>` tag. Next, we have a `<view>` tag with a title parameter, which is set to the name we want to use in the back end when referring to this view. The `<message>` tag is placed within this tag and contains the description of the view seen on mouse over. Go to the back end and select **Menus | Main Menu**, then click **New**. After selecting **Restaurant Reviews** as the menu type, you should be presented with a screen like the following:



Clicking on **Default Layout** under **All** will give you a configuration screen including the menu item parameters box as seen in the following figure:



Currency Symbol $

Defining the currency symbol in the configuration display in the front end will require some modifications to the display class controlling individual reviews. Open `/components/com_reviews/views/review/view.html.php` and make the following highlighted modifications:

```
class ReviewViewReview extends JView
{
  function display($tpl = null)
  {
    global $option, $mainframe;
    $model = &$this->getModel();
    $user =& JFactory::getUser();
    $review = $model->getReview();
    $comments = $model->getComments();
    $pathway =& $mainframe->getPathWay();
```

```
        $backlink = JRoute::_('index.php?option=' . $option);
        $menu =& JMenu::getInstance();
        $item = $menu->getActive();
        $params =& $menu->getParams($item->id);
        $currency = $params->get('currency_symbol', '$');
        $review->review_date = JHTML::Date($review->review_date);
        if($review->smoking == 1)
        {
          $review->smoking = "Yes";
        }
        else
        {
          $review->smoking = "No";
        }
        for($i = 0; $i < count($comments); $i++)
        {
          $row =& $comments[$i];
          $row->comment_date = JHTML::Date($row->comment_date);
        }
        $pathway->addItem($review->name, '');
        $this->assignRef('review', $review);
        $this->assignRef('comments', $comments);
        $this->assignRef('backlink', $backlink);
        $this->assignRef('Itemid', $Itemid);
        $this->assignRef('option', $option);
        $this->assignRef('name', $user->_table->name);
        $this->assignRef('currency', $currency);
        parent::display($tpl);
      }
    }
```

To include the currency symbol set in the configuration, we need to load the parameters. Since the parameters are saved with the menu item pointing to the component, we first use the `getInstance()` member function of `JMenu` to get a reference to an object we set in `$menu`. We then call the `getActive()` member function of `$menu` to get a reference to the current menu item. Finally, we get a reference to the parameters object by calling the `getParams()` member function of `$menu` and pass in the `id` member variable of `$item`. As with other parameters, we use the `get()` member function to set `$currency` with the value of `currency_symbol`, defaulting to `$` if none is specified. This variable is then assigned by reference to the `ReviewViewReview` object.

The display template itself will also need an adjustment. Open `/components/com_reviews/views/review/tmpl/default.php` and replace the `$` symbol on the highlighted line with this echo command:

```
<p class="contentheading">
  <?php echo $this->review->name; ?>
</p>
<p class="createdate">
  <?php echo $this->review->review_date; ?>
</p>
<p>
  <?php echo $this->review->quicktake; ?>
</p>
<p><strong>Address:</strong> <?php echo $this->review->address; ?>
</p>
<p><strong>Cuisine:</strong> <?php echo $this->review->cuisine; ?>
</p>
<p><strong>Average dinner price:</strong>
  <?php echo $this->currency, $this->review->avg_dinner_price; ?>
</p>
<p><strong>Credit cards:</strong>
  <?php echo $this->review->credit_cards; ?>
</p>
<p><strong>Reservations:</strong>
  <?php echo $this->review->reservations; ?>
</p>
<p><strong>Smoking:</strong>
  <?php echo $this->review->smoking ?>
</p>
<p>
  <?php echo $this->review->review; ?>
</p>
<p><em>Notes:</em>
  <?php echo $this->review->notes; ?>
</p>
<a href="<?php echo $this->backlink; ?>">&lt; return to the
                              reviews</a>
<?php if(count($this->comments)) : ?>
  <br /><br />
  <?php foreach($this->comments as $comment): ?>
  <p><strong><?php echo $comment->full_name; ?></strong> <em><?php
                 echo $comment->comment_date; ?></em></p>
  <p><?php echo $comment->comment_text; ?></p>
  <?php endforeach; ?>
```

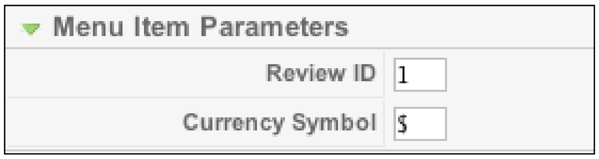
```
<?php endif; ?>
<br /><br />
<?php echo $this->loadTemplate('form'); ?>
```

As with the previously added variables, we output the `currency` member variable of the view object. We then immediately follow this with the restaurant's average dinner price.

In addition to linking to all of the reviews in a directory style format, we want to be able to link to individual reviews. When linking to these individual reviews, we want the same control over the currency symbol that we now have for reviews as a whole. To do so, we need to create `metadata.xml` in `/components/com_reviews/views/review` with the following code:

```
<?xml version="1.0" encoding="utf8"?>
<metadata>
  <view title="Single Review">
    <message>
      <![CDATA[Shows individual reviews.]]>
    </message>
  </view>
  <params>
    <param name="id" type="text" size="3" default=""
               label="Review ID" description="Enter the ID
               of the review to be displayed." />
    <param name="currency_symbol" type="text" size="3"
               default="$" label="Currency Symbol"
               description="Enter the currency symbol to
               be used for average dinner prices." />
  </params>
</metadata>
```

This XML configuration is similar to the one we used for the `All` view. Besides the labeling, the main difference is that we have an additional parameter where the admin can add the ID of a specific review to be displayed. When creating a new menu link using the `Single Review` link under **Restaurant Reviews**, you should now get a parameters box that looks like the following image. Get the **Review ID** for the review you wish to link to and enter it in.


Menu Item Parameters

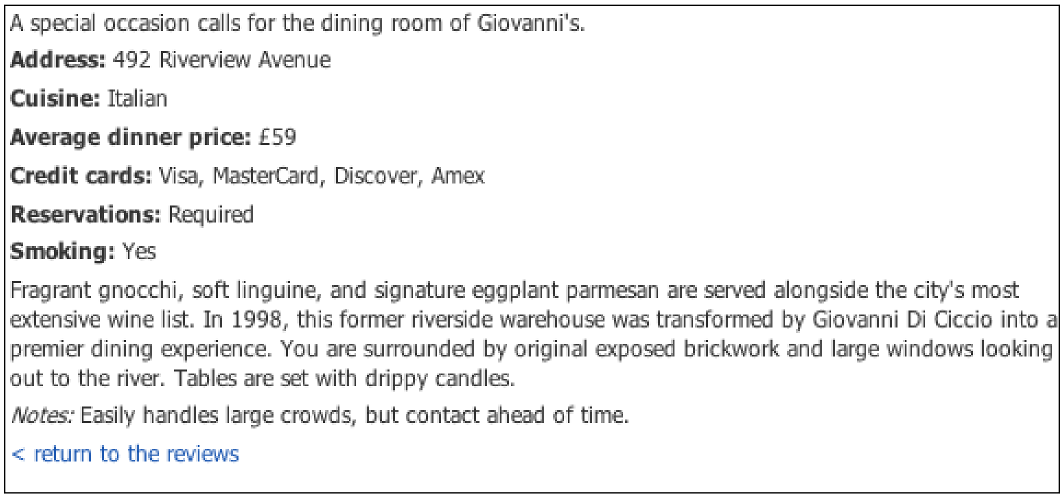
Review ID	1
Currency Symbol	$

Although we will not have to make any modifications to the display class, we will have to modify the model constructor to detect whether or not an ID has been set in the menu item parameters. Open `/components/com_reviews/models/review.php` and replace the `__construct()` function with the following code:

```
function __construct()
{
  parent::__construct();
  $params =& JSiteHelper::getMenuParams();
  $id = $params->get('id', 0);
  if(!$id)
  {
    $id = JRequest::getVar('id', '');
  }
  $this->_id = $id;
}
```

As in the previous version of `__construct()`, we call the `JModel` parent constructor as we only want to add some steps to the class initialization process, not replace it. We then get a reference to the current menu item's parameters by using the `getMenuParams()` member function of `JSiteHelper`. We capture the value of the `id` parameter in `$id`, defaulting to `0` if `id` is not set. The value of `$id` is then tested; if it's set to `0`, we attempt to get `id` from the request variables, defaulting to `null` if it does not exist. Finally, we set the object's `_id` member variable to the value of `$id`.

After saving the model, go to the front end and follow the link you created to the individual review. Your screen should look something like the following:



A special occasion calls for the dining room of Giovanni's.

Address: 492 Riverview Avenue

Cuisine: Italian

Average dinner price: £59

Credit cards: Visa, MasterCard, Discover, Amex

Reservations: Required

Smoking: Yes

Fragrant gnocchi, soft linguine, and signature eggplant parmesan are served alongside the city's most extensive wine list. In 1998, this former riverside warehouse was transformed by Giovanni Di Ciccio into a premier dining experience. You are surrounded by original exposed brickwork and large windows looking out to the river. Tables are set with drippy candles.

Notes: Easily handles large crowds, but contact ahead of time.

< return to the reviews

Summary

Through XML configuration files, we've been able to add several options without creating separate tables to hold the values. Parameters have been added to modules, plug-ins, and component views with minimally invasive code. Site administrators are now able to use familiar controls to manage these options.

9
Packing Everything Together

Our restaurant reviewers are now quite satisfied with the development of the site; satisfied enough to recommend it to their restaurant reviewing colleagues in other cities. Our emails are now flooded with requests to produce similar sites. Instead of producing each individual website, we will package the module, component, and plug-ins so they can be sold! Our packaging process will cover the following tasks:

- Listing all files
- Packaging the component
- Creating back-end menu options
- Including SQL queries
- Extra installation scripts
- Distribution

Listing All Files

To create the installation packages for our elements, we will start with the XML files we previously created to hold the configuration parameters. All the three extension types require you to list all the files in the package. The installer will not copy over files in the package that are not listed in the XML file.

Packaging the Module

For our module, take the existing `mod_reviews.xml` file and add the following highlighted code:

```
<?xml version="1.0" encoding="utf-8"?>
<install type="module" version="1.5">
  <name>Restaurant Reviews</name>
  <author>Sumptuous Software</author>
  <creationDate>January 2007</creationDate>
  <copyright>(C) 2007</copyright>
  <license>Commercial</license>
  <authorEmail>support@packtpub.com</authorEmail>
  <authorUrl>www.packtpub.com</authorUrl>
  <version>1.0</version>
  <description>A module for promoting restaurant
           reviews.</description>
  <files>
    <filename module="mod_reviews">mod_reviews.php</filename>
    <filename>helper.php</filename>
    <filename>tmpl/_review.php</filename>
    <filename>tmpl/bulleted.php</filename>
    <filename>tmpl/default.php</filename>
  </files>
  <params>
    <param name="random" type="radio" default="0"
          label="Randomize" description="Show random reviews">
      <option value="0">No</option>
      <option value="1">Yes</option>
    </param>
    <param name="@spacer" type="spacer" default=""
          label="" description="" />
    <param name="items" type="text" default="1"
          label="Display #" description="Number of
          reviews to display" />
    <param name="style" type="list" default="default"
          label="Display style" description="The style
          to use for displaying the reviews.">
      <option value="default">Flat</option>
      <option value="bulleted">Bulleted</option>
    </param>
  </params>
</install>
```

Each file is listed in a `<filename>` element and all of these are enclosed in a `<files>` element. For the `mod_reviews.php` file, we give the `<filename>` element a parameter of `module` set to `mod_reviews`. This creates the directory `mod_reviews` in the `modules` directory and also registers our module in the database. By doing this, we automate the steps while writing the first module.

Packaging Plug-ins

The adjustments to the XML files for the plug-ins are very similar. Open `/plugins/conent/reviews.xml` and make the following changes:

```
<?xml version="1.0" encoding="utf-8"?>
<install version="1.5" type="plugin" group="content">
  <name>Content - Restaurant Review Links</name>
  <author>Sumptuous Software</author>
  <creationDate>January 2007</creationDate>
  <copyright>(C) 2007</copyright>
  <license>Commercial</license>
  <authorEmail>support@packtpub.com</authorEmail>
  <authorUrl>www.packtpub.com</authorUrl>
  <version>1.0</version>
  <description>Searches for titles of restaurants in articles
           and turns them into review links.</description>
  <files>
    <filename plugin="reviews">reviews.php</filename>
  </files>
  <params>
    <param name="linkcode" type="textarea" default="" rows="5"
          cols="40" label="Custom Link Code" description="By
          using {link} and {title}, you can generate custom
          HTML output that includes the URL and review title
          respectively." />
  </params>
</install>
```

On the opening `<install>` tag, we add the `group` parameter and set it to `content`. This ensures the plug-in is added to the correct directory. For the single file of code, we have the parameter `plugin` set to the plug-in name, which is used along with the group in the database to identify it.

The process is identical for the `reviewinfo` plug-in:

```
<?xml version="1.0" encoding="utf-8"?>
<install version="1.5" type="plugin" group="content">
  <name>Content - Review Information</name>
  <author>Sumptuous Software</author>
  <creationDate>January 2007</creationDate>
  <copyright>(C) 2007</copyright>
  <license>Commercial</license>
  <authorEmail>support@packtpub.com</authorEmail>
  <authorUrl>www.packtpub.com</authorUrl>
  <version>1.0</version>
  <description>Turns {reviewinfo Name of your restaurant} into a table
with the review's essential details.</description>
  <files>
    <filename plugin="reviewinfo">reviewinfo.php</filename>
  </files>
  <params>
    <param name="address" type="radio" default="1"
        label="Display Address?" description="Toggles the
        display of the address in summaries.">
      <option value="1">Yes</option>
      <option value="0">No</option>
    </param>
    <param name="price_range" type="radio" default="1"
        label="Display Price Range?" description="Toggles the
        display of the price range in summaries.">
      <option value="1">Yes</option>
      <option value="0">No</option>
    </param>
    <param name="reservations" type="radio" default="1"
        label="Display Reservations?" description="Toggles the
        display of reservation policy in summaries.">
      <option value="1">Yes</option>
      <option value="0">No</option>
    </param>
    <param name="smoking" type="radio" default="1"
        label="Display Smoking?" description="Toggles the
        display of smoking policy in summaries.">
      <option value="1">Yes</option>
      <option value="0">No</option>
    </param>
  </params>
</install>
```

The changes to the XML file for the `search` plug-in are similar to the first two, except that here you set the group parameter in `<install>` to content. Notice that we are using the same name for a plug-in in the `search` group as in the `content` group. This is possible because of the manner in which XML is written.

```
<?xml version="1.0" encoding="utf-8"?>
<install version="1.5" type="plugin" group="search">
  <name>Search - Restaurant Reviews</name>
  <author>Sumptuous Software</author>
  <creationDate>January 2007</creationDate>
  <copyright>(C) 2007</copyright>
  <license>Commercial</license>
  <authorEmail>support@packtpub.com</authorEmail>
  <authorUrl>www.packtpub.com</authorUrl>
  <version>1.0</version>
  <description>Allows Searching of Restaurant Reviews</description>
  <files>
    <filename plugin="reviews">reviews.php</filename>
  </files>
  <params>
    <param name="search_limit" type="text" size="5" default="50"
        label="Search Limit" description="Number of Search items
        to return"/>
  </params>
</install>
```

Packaging the Component

Although preparing modules and plug-ins mainly involves listing the files, components need some extra attention. Components are typically used to manage records in the database, so queries to add the accompanying tables are necessary. We will require a link to the component back end. Finally, we may wish to run some additional set-up code just after installation or a clean-up script when the component is removed. For the moment, create `reviews.xml` in `/components/com_reviews` and add the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<install type="component" version="1.5.0">
  <name>Reviews</name>
  <author>Sumptuous Software</author>
  <creationDate>January 2007</creationDate>
  <copyright>(C) 2007</copyright>
  <authorEmail>support@packtpub.com</authorEmail>
  <authorUrl>www.packtpub.com</authorUrl>
  <version>1.5.0</version>
```

```
     <license>Commercial</license>
     <description>A component for writing and managing
             restaurant reviews.</description>
     <installfile>install.reviews.php</installfile>
     <uninstallfile>uninstall.reviews.php</uninstallfile>
     <install>
       <sql>
        <file driver="mysql" charset="utf8">install.mysql.sql</file>
       </sql>
     </install>
     <uninstall>
       <sql>
         <file driver="mysql" charset="utf8">uninstall.mysql.sql</file>
       </sql>
     </uninstall>
     <files>
       <filename>controller.php</filename>
       <filename>reviews.html.php</filename>
       <filename>reviews.php</filename>
       <filename>router.php</filename>
       <filename>models/review.php</filename>
       <filename>models/all.php</filename>
       <filename>views/all/view.html.php</filename>
       <filename>views/all/tmpl/default.php</filename>
       <filename>views/review/view.html.php</filename>
       <filename>views/review/tmpl/default.php</filename>
       <filename>views/review/tmpl/default_form.php</filename>
     </files>
     <administration>
       <menu>Restaurant Reviews</menu>
       <submenu>
         <menu link="option=com_reviews">Manage Reviews</menu>
         <menu task="comments">Manage Comments</menu>
       </submenu>
       <files folder="admin">
         <filename>install.mysql.sql</filename>
         <filename>uninstall.mysql.sql</filename>
         <filename>admin.reviews.html.php</filename>
         <filename>admin.reviews.php</filename>
         <filename>controller.php</filename>
         <filename>tables/comment.php</filename>
         <filename>tables/review.php</filename>
         <filename>toolbar.reviews.html.php</filename>
         <filename>toolbar.reviews.php</filename>
       </files>
     </administration>
   </install>
```

What has Changed from Joomla! 1.0?

For the most part, XML component installation files for Joomla! 1.5 are similar to the ones used in 1.0. For the installation and uninstallation queries, the SQL is now migrated to external files, with the flexibility of including different SQL files for different database types. Also, the back-end code is now sorted into a separate folder that you can specify in the `folder` parameter of the `<files>` tag in the `<administration>` section. This helps to avoid filename conflicts.

As with modules and plug-ins, we list all of the files related to the extension. However, with components, we have back-end files as well as front-end files. The back-end files are placed within the `<administration>` tag in a `<files>` tag where the `folder` attribute is set to `admin`. The files enclosed within `<installfile>` and `<uninstallfile>` tags are used to identify the custom installation and uninstallation files that we will create in a moment. Although we will locate these files in `/components/com_reviews`, they will be moved to `/administrator/components/com_reviews` upon installation.

Beneath the tags for custom installation and uninstallation files are the `<install>` and `<uninstall>` tags. Within these tags are the `<sql>` and `<file>` tags. These are used to add SQL queries to the installation and uninstallation processes. Since Joomla! supports different database types, you can include a different file for each type (we will only create one for MySQL). Note that both the `install.mysql.sql` and `uninstall.mysql.sql` files are listed within the `<install>` and `<uninstall>` tags, as well as the `<files>` tag within the `<administration>` tag. If these files are not also listed in the `<administration>` section, they will not be copied on installation and the queries will consequently not be run. This is in contrast to the files listed in the `<installfile>` and `<uninstallfile>` folders.

Including SQL Queries

To add the tables that we need for managing the reviews, some SQL queries should be run. To do this, we will add the queries to some files that will be run when the component is installed and uninstalled. Create the file `install.mysql.sql` in `/administrator/components/com_reviews` and paste the following queries:

```
CREATE TABLE IF NOT EXISTS '#__reviews' (
  'id' int(11) NOT NULL auto_increment,
  'name' varchar(255) NOT NULL,
  'address' varchar(255) NOT NULL,
  'reservations' varchar(31) NOT NULL,
  'quicktake' text NOT NULL,
  'review' text NOT NULL,
  'notes' text NOT NULL,
  'smoking' tinyint(1) NOT NULL default '0',
  'credit_cards' varchar(255) NOT NULL,
  'cuisine' varchar(31) NOT NULL,
  'avg_dinner_price' tinyint(3) NOT NULL default '0',
  'review_date' datetime NOT NULL,
  'published' tinyint(1) NOT NULL default '0',
  PRIMARY KEY  ('id')
);

CREATE TABLE IF NOT EXISTS '#__reviews_comments' (
  'id' int(11) NOT NULL auto_increment,
  'review_id' int(11) NOT NULL,
  'user_id' int(11) NOT NULL,
  'full_name' varchar(50) NOT NULL,
  'comment_date' datetime NOT NULL,
  'comment_text' text NOT NULL,
  PRIMARY KEY  ('id')
);
```

There are two differences from the originals encountered earlier. First, we've added the additional qualifier `IF NOT EXISTS`. If someone has problems uninstalling the component or already has these tables otherwise (perhaps from a backup), this will prevent an error from occurring. Also, in these queries, we're using the `#_` table prefix notation to be replaced with the one on the host Joomla! System.

In addition to the installation SQL, we want to provide an uninstallation SQL script that will remove the tables so that no trace of the component is left. Create the file `uninstall.mysql.sql` in `/administrator/components/com_reviews` and add the following code:

```
DROP TABLE #__reviews;
DROP TABLE #__reviews_comments;
```

The code from both of these files will be used as we've defined them within the `<install>` and `<uninstall>` tags we added to the XML file.

Creating Back-End Menu Items

Within the `<administration>` tags in the XML file, we define the items found under the **Components** menu item in the back end. If we were only managing one type of record, the following piece of XML would be sufficient for linking to the back end:

```
<menu>Restaurant Reviews</menu>
```

However, our component manages both reviews and comments. To handle this, we want the `Restaurant Reviews` item to expand into two submenu items. In the following XML, we enclose the menu items in a `<submenu>` tag. The first item uses `link` to define a hard link to `index2.php?option=com_reviews`, while the second uses `task` to form a link to `index2.php?option=com_reviews&task=comments`.

```
<submenu>
  <menu link="option=com_reviews">Manage Reviews</menu>
  <menu task="comments">Manage Comments</menu>
</submenu>
```

Extra Installation Scripts

When installing a component, Joomla! displays a standard success message along with the description found in the XML file. A generic uninstallation message is also generated upon removal. We can override both these with custom code. Create the `install.reviews.php` file in `/components/com_reviews` and enter the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
function com_install()
{
  ?>
  <div class="header">Congratulations, Restaurant Reviews is
             ready to roll!</div>
  <p>
  Congratulations on not only purchasing, but also installing
             Restaurant Reviews! Undoubtedly, you are about to
             embark on many joyous hours of authoring and
             organizing all of the hot spots for your city.
             To get started, navigate to Components, Restaurant
             Reviews, Manage Reviews and click the "New" button
             at the right-hand corner of the screen. Also, be
             sure to install the accompanying plugins and module
             to promote your reviews throughout the website!
  </p>
```

```
    <?php
  }
  ?>
```

For uninstallation, create the `uninstall.reviews.php` file in `/components/com_reviews` containing the following code:

```
<?php
defined( '_JEXEC' ) or die( 'Restricted access' );
function com_uninstall()
{
  ?>
  <div class="header">The reviews are now removed from
              your system.</div>
  <p>
  We're sorry to see you go! To completely remove the
              software from your system, be sure to also
              uninstall the plugins and module.
  </p>
  <?php
}
?>
```

These scripts are referenced by the XML in the following two lines we added earlier below the `<description>` tag:

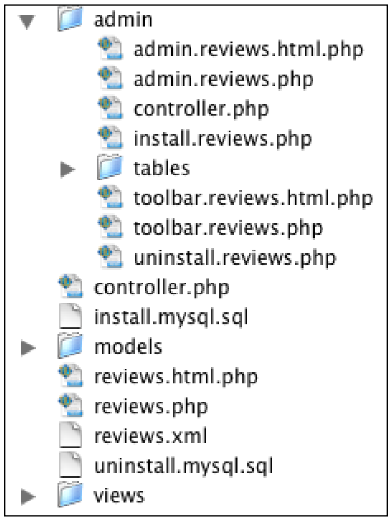
```
<installfile>install.reviews.php</installfile>
<uninstallfile>uninstall.reviews.php</uninstallfile>
```

Joomla! will load the `install.reviews.php` file on installation and `uninstall.reviews.php` on removal, but will call the functions `com_install()` and `com_uninstall()` respectively. You can use these functions to do more than simply display messages. The `com_install()` function is called just after the installation process is complete, so it can be used to guide users through first-time configuration. Likewise, the `com_uninstall()` function is called just before the component is removed; any output generated will be buffered and displayed after the component is removed. If `com_install()` or `com_uninstall()` return false, the process is rolled back. This can be used to prevent components from being installed when the target system does not meet the minimum requirements. It can also be used to prevent the removal of a component that published menu links point to.

Distribution

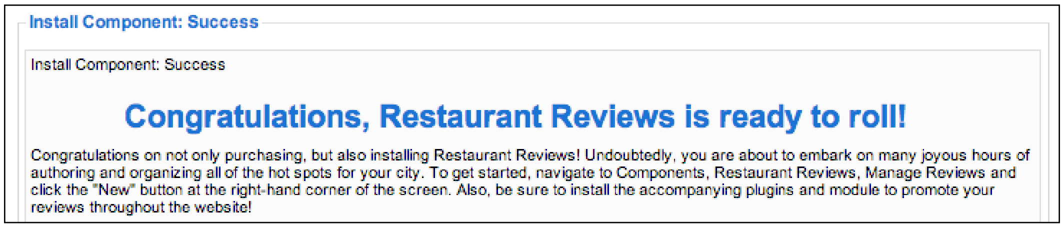
We now have all the files we need to package our extensions. For the module, put all the files and folders into a `.zip` archive and place them in `/modules/mod_review`. For the plug-ins, create three separate `.zip` archives: first for the review information, second for the review links, and third for the review searches. Each of these archives should contain the `.php` and `.xml` file for the corresponding plug-in.

The component needs a little extra attention. Since both the front end and back end contain a file named `controller.php`, we need to place one of the set of files in a separate folder within the archive. Since the file listing has the administrative files designated as being in the `admin` folder, this is the one we will create. The structure of your component archive should look like the following:



```
▼ admin
      admin.reviews.html.php
      admin.reviews.php
      controller.php
      install.reviews.php
    ▶ tables
      toolbar.reviews.html.php
      toolbar.reviews.php
      uninstall.reviews.php
  controller.php
  install.mysql.sql
▶ models
  reviews.html.php
  reviews.php
  reviews.xml
  uninstall.mysql.sql
▶ views
```

After creating these five archives, all the code created in this book will be ready for installation on any Joomla! system. Set up a clean installation of Joomla! (apart from the one you used to develop the component) and install the component by going to **Extensions | Install/Uninstall**, then use the **Upload Package File form** to upload the `.zip` archive containing the reviews component. If everything works correctly, you should see the following screen:



Install Component: Success

Install Component: Success

Congratulations, Restaurant Reviews is ready to roll!

Congratulations on not only purchasing, but also installing Restaurant Reviews! Undoubtedly, you are about to embark on many joyous hours of authoring and organizing all of the hot spots for your city. To get started, navigate to Components, Restaurant Reviews, Manage Reviews and click the "New" button at the right-hand corner of the screen. Also, be sure to install the accompanying plugins and module to promote your reviews throughout the website!

Summary

We now have several `.zip` files ready to go with everything necessary to set the Restaurant Reviews system up on another website. We've spared our end users from confusing queries: they simply upload the files through the **Extension manager** and start writing reviews! This is made possible through the XML configuration file defining the scripts to run, queries to add, and files to copy on installation.

Index