Mastering Joomla! 1.5
Extension and Framework Development

James Kennard

Chapter No. 3
"The Database"
In this package, you will find:
A Biography of the author of the book
A preview chapter from the book, Chapter NO. 3 "The Database"
A synopsis of the book’s content
Information on where to buy this book

About the Author

James Kennard is a computer programmer with a particular interest in web-based services. His interests in Joomla! started as a result of an internal assignment at work when he was tasked with identifying suitable web systems to host a number of intranet and Internet applications.

James currently maintains one open-source Joomla! component, which has been translated into over fifteen languages. Examples of his work can be found on his personal website: www.webamoeba.co.uk.
Mastering Joomla! 1.5
Extension and Framework Development

This book will guide you through the complexities of implementing components, modules, and plugins in Joomla! 1.5. It provides useful reference material that explains many of the advanced design features and classes available in Joomla! 1.5.

Joomla! is one of the world's top open-source content management systems. The main sources of the PHP MySQL application's success are its comprehensive extension libraries, which extend Joomla! far beyond content management, and its very active forums where one can easily tap into the knowledge of other Joomla! users, administrators, and developers.

The architecture of the latest version of Joomla! differs in many ways from previous versions. Resultantly backward-compatibility with some extensions has been broken; the race is on for developers to update their skills in order to rectify the problems and start building new extensions. Perhaps the most important of the changes is the reorganization and classification of files and classes. This change encourages but does not force developers to use the Joomla! libraries consistently between extensions.

What This Book Covers

Chapter 1 deals with the history of Joomla! and gives an overview of the technology in general.

Chapter 2 covers the process from request to response and also talks about directory and URI structure along with a brief description of libraries. It also introduces a number of common classes, variables, and constants that are used frequently when creating Joomla! extensions.

Chapter 3 deals with the database. It talks about extending the database, conventions for the database schema, and common fields. Then the focus moves on to storing data common types of data in standard fields and dealing with multilingual requirements. We then cover querying the database and getting results. Next, the chapter explores how to manipulate common field types. The chapter concludes with a brief description of the JTable. The JTable is used to display and edit regular two-dimensional tables of cells. The JTable has many facilities that make it possible to customize its rendering and editing but provides defaults for these features so that simple tables can be set up easily.

Chapter 4 is about designing components. It starts with the structure and a basic design of a component using the MVC design pattern. Then we learn configuring the component and its various elements and parameters. The chapter finishes by discussing component packaging and the various install and uninstall files.
Chapter 5 covers designing modules. It explains standalone modules, module settings, frontend and backend modules, and modules and components working together. Then we talk about using templates and packaging the modules.

Chapter 6 deals with designing plugins. It initially deals with listeners/observers and then the various plugin groups like authentication, content editors, search, and others. Then comes loading, translating, and using plugins as libraries. Finally it deals with, plugin settings and how to package plugins.

Chapter 7 is all about designing extensions. Here, we start with helper classes then cover building and using getInstance() methods. Then we cover the registry along with saving and loading registry values. Towards the end of the chapter, we explain the User, Session, Browser and the assets.

Chapter 8 explains ways to render output and how to maintain consistency throughout. It starts with the joomla.html library and then continues to describe how to build component HTML layouts. Then it discusses how to output the backend of a component. The chapter ends with the details of itemized data and pagination.

Chapter 9 deals with customizing the page. We cover things like modifying the document and translating, along with a brief explanation of using JavaScript effects from the mootools library, which is included in Joomla!.

Chapter 10 explores some of the Joomla! APIs, specifically in relation to web services. We also discuss some of the more common web services and take a more in-depth look at the Yahoo! Search API. The chapter finishes by describing how we can create our own web services using plugins.

Chapter 11 provides an introduction to handling and throwing errors, warnings, and notices. Further, it talks about building secure Joomla! extensions. It also describes a number of common mistakes made when coding with Joomla! and explains how to avoid them.

Chapter 12 explains various utilities and useful classes like dates, arrays, tree structures, and others.

The Appendix details the more common Joomla! classes. It also provides information on how to handle the ever-useful JParameter object. The appendix ends with a description of the Joomla! settings in relation to the registry/config.
The Database

This chapter details the role of the database in Joomla!. It defines some standard rules we need to abide by. It explains different ways in which we can query the database. It also briefly covers the ADOdb emulation that is available for developers wanting to port existing applications.

Joomla! is currently designed to use the MySQL database. However, the architecture does allow for the implementation of other database drivers. There is some uncertainty surrounding the issue of supporting other databases, because of the usage in queries of functions and syntax that are specific to MySQL.

The Core Database

Much of the data we see in Joomla! is stored in the database. A base installation has over thirty tables. Some of these are related to core extensions and others to the inner workings of Joomla!.

There is an official database schema, which describes the tables created during the installation. For more information, please refer to: http://dev.joomla.org/component/option,com_jd-wiki/Itemid,31/id, guidelines:database/.


We access the Joomla! database using the global JDatabase object. The JDatabase class is an abstract class, which is extended by different database drivers. There are currently only two database drivers included in the Joomla! core, MySQL and MySQLi. We access the global JDatabase object using JFactory:

```php
$db =& JFactory::getDBO();
```
The Database

Extending the Database
When we create extensions, we generally want to store data in some form. If we are using the database, it is important to extend it in the correct way. More information on extending the database with components is available in Chapter 4.

Table Prefix
All database tables have a prefix, normally `jos_`, which helps in using a single database for multiple Joomla! installations. When we write SQL queries, to accommodate the variable table prefix, we use a symbolic prefix that is substituted with the actual prefix at run time. Normally the symbolic prefix is `#__`, but we can specify an alternative prefix if we want to.

Schema Conventions
When we create tables for our extensions, we must follow some standard conventions. The most important of these is the name of the table. All tables must use the table prefix and should start with name of the extension. If the table is storing a specific entity, add the plural of the entity name to the end of the table name separated by an underscore. For example, an items table for the extension 'My Extension' would be called `#__myExtension_items`.

Table field names should all be lowercase and use underscore word separators; you should avoid using underscores if they are not necessary. For example, you can name an email address field as `email`. If you had a primary and a secondary email field, you could call them `email` and `email_secondary`; there is no reason to name the primary email address `email_primary`.

If you are using a primary key record ID, you should call the field `id`, make it of type `integer auto_increment`, and disallow `null`. Doing this will allow you to use the Joomla! framework more effectively.

Common Fields
We may use some common fields in our tables. Using these fields will enable us to take advantage of the Joomla! framework. We will discuss how to implement and manipulate these fields, using the JTable class, later in this chapter.
Chapter 3

Publishing
We use publishing to determine whether to display data. Joomla! uses a special field called published, of type tinyint(1); 0 = not published, 1 = published.

Hits
If we want to keep track of the number of times a record has been viewed, we can use the special field hits, of type integer and with the default value 0.

Checking Out
To prevent more than one user trying to edit one record at a time we can check out records (a form of software record locking). We use two fields to do this, checked_out and checked_out_time. checked_out, of type integer, holds the ID of the user that has checked out the record. checked_out_time, of type datetime, holds the date and time when the record was checked out. A null date and a user ID of 0 is recorded if the record is not checked out.

Ordering
We often want to allow administrators the ability to choose the order in which items appear. The ordering field, of type integer, can be used to number records sequentially to determine the order in which they are displayed. This field does not need to be unique and can be used in conjunction with WHERE clauses to form ordering groups.

Parameter Fields
We use a parameter field, a TEXT field normally named params, to store additional information about records; this is often used to store data that determines how a record will be displayed. The data held in these fields is encoded as INI strings (which we handle using the JParameter class). Before using a parameter field, we should carefully consider the data we intend to store in the field. Data should only be stored in a parameter field if all of the following criteria are true:

- Not used for sorting records
- Not used in searches
- Only exists for some records
- Not part of a database relationship
The Database

Schema Example
Imagine we have an extension called 'My Extension' and an entity called foobar. The name of the table is `#__myextension_foobars`. This schema describes the table:

<table>
<thead>
<tr>
<th>Field</th>
<th>Datatype</th>
<th>NOT NULL</th>
<th>AUTO INC</th>
<th>UNSIGNED</th>
<th>DEFAULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>INTEGER</td>
<td>✓</td>
<td></td>
<td></td>
<td>NULL</td>
</tr>
<tr>
<td>content</td>
<td>TEXT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checked_out</td>
<td>INTEGER</td>
<td>✓</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>checked_out_time</td>
<td>DATETIME</td>
<td>✓</td>
<td></td>
<td></td>
<td>0000-00-00 00:00:00</td>
</tr>
<tr>
<td>params</td>
<td>TEXT</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ordering</td>
<td>INTEGER</td>
<td>✓</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>hits</td>
<td>INTEGER</td>
<td>✓</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>published</td>
<td>TINYINT(1)</td>
<td>✓</td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

This table uses all of the common fields and uses an auto-incrementing primary key ID field. When we come to define our own tables we must ensure that we use the correct data types and NOT NULL, AUTO INC, UNSIGNED and DEFAULT values.

The SQL displayed below will create the table described in the above schema:

```
CREATE TABLE `#__myextension_foobars` (
    'id' INTEGER UNSIGNED NOT NULL DEFAULT NULL AUTO_INCREMENT,
    'content' TEXT NOT NULL DEFAULT '',
    'checked_out' INTEGER UNSIGNED NOT NULL DEFAULT 0,
    'checked_out_time' DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00',
    'params' TEXT NOT NULL DEFAULT '',
    'ordering' INTEGER UNSIGNED NOT NULL DEFAULT 0,
    'hits' INTEGER UNSIGNED NOT NULL DEFAULT 0,
    'published' INTEGER UNSIGNED NOT NULL DEFAULT 0,
    PRIMARY KEY('id')
) CHARACTER SET 'utf8' COLLATE 'utf8_general_ci';
```

Date Fields
We regularly use datetime fields to record the date and time at which an action has taken place. When we use these fields, it is important that we are aware of the effect of time zones. All dates and times should be recorded in UTC+0 (GMT / Z).
When we come to display dates and times we can use the JDate class, described in Chapter 12. The JDate class allows us to easily parse dates, output them in different formats, and apply UTC time-zone offsets.

For more information about time zones, please refer to http://www.timeanddate.com.

We often use parsers before we display data to make the data safe or to apply formatting to the data. We need to be careful how we store data that is going to be parsed. If the data is ever going to be edited, we must store the data in its RAW state. If the data is going to be edited extremely rarely and if the parsing is reversible, we may want to consider building a 'reverse-parser'. This way we can store the data in its parsed format, eradicating the need for parsing when we view the data and reducing the load on the server. Another option available to us is to store the data in both formats. This way we only have to parse data when we save it.

Dealing with Multilingual Requirements

In the previous chapter we discussed Joomla!’s use of the Unicode character set using UTF-8 encoding. Unlike ASCII and ANSII, Unicode is a multi-byte character set; it uses more than eight bits (one byte) per character. When we use UTF-8 encoding, character byte lengths vary.

Unfortunately, MySQL versions prior to 4.1.2 assume that characters are always eight bits (one byte), which poses some problems. To combat the issue when installing extensions we have the ability to define different SQL files for servers, that do and do not support UTF-8.

In MySQL servers that do not support UTF-8, when we create fields, which define a character length, we are actually defining the length in bytes. Therefore, if we try to store UTF-8 characters that are longer than one byte, we may exceed the size of the field. To combat this, we increase the length of fields to try to accommodate UTF-8 strings. For example, a varchar(20) field becomes a varchar(60) field. We triple the size of fields because, although UTF-8 characters can be more than three bytes, the majority of common characters are a maximum of three bytes.

This poses another issue, if we use a varchar(100) field, scaling it up for a MySQL server, which does not support UTF-8, we would have to define it as a varchar(300) field. We cannot do this because varchar fields have a maximum size of 255. The next step is slightly more drastic. We must redefine the field type so as it will accommodate at least three hundred bytes. Therefore, a varchar(100) field becomes a text field.
As an example, the core `#__content` table includes a field named `title`. For MySQL servers that support UTF-8, the field is defined as:

    'title' varchar(255) NOT NULL default ''

For MySQL servers that do not support UTF-8, the field is defined as:

    'title' text NOT NULL default ''

We should also be aware that using a version of MySQL that does not support UTF-8 would affect the MySQL string handling functions. For example ordering by a string field may yield unexpected results. While we can overcome this using post-processing in our scripts using the JString class, the recommended resolution is to upgrade to the latest version of MySQL.

### Querying the Database

When we perform a query, we tell the global JDatabase object the query that we want to execute. We do this using the `setQuery()` method; this does not perform the query.

```php
$db = & JFactory::getDBO();
$result = $db->setQuery($query);
```

Once we have set the query we want to perform, we use the `query()` method to execute the query. This is similar to using the PHP function `mysql_query()`. If the query is successful and is a `SELECT`, `SHOW`, `DESCRIBE`, or `EXPLAIN` query, a resource will be returned. If the query is successful, and is not one of the above query types, true will be returned. If the query fails, false will be returned.

```php
$db = & JFactory::getDBO();
if (!$result = $db->setQuery($query))
{
    // handle failed query
    // use $table->getError() for more information
}
```

### Writing Queries

There are some rules we need to be aware of when we build database queries.

- Use the `#__` symbolic prefix at the start of all table names.
- Use the `nameQuote()` method to encapsulate named query elements.
- Use the `Quote()` method to encapsulate values.
The symbolic prefix guarantees that we use the correct prefix for the current Joomla! installation; an alternative symbolic prefix to #__ can be used if necessary. _nameQuote()_ ensures that named elements are encapsulated in the correct delimiters. _Quote()_ ensures that values are encapsulated in the correct delimiters. This example demonstrates the use of all of these rules.

```php
$db = JFactory::getDBO();
$query = 'SELECT * FROM ' . $db->nameQuote('#__test') . ' WHERE ' . $db->nameQuote('name') . ' = ' . $db->Quote('Some Name');
```

If we were using a MySQL or MySQLi database driver, $query would equal the following:

```
SELECT * FROM 'jos_test' WHERE 'name' = "Some Name";
```

### Getting Results

We could use the _query()_ method and process the resultant resource. However, it is far easier to use one of the other JDatabase methods, which will get the results from a query in a number of different formats.

To help explain each of the methods we will use a sample table called #__test. The table has two fields, _id_, an auto-increment primary key, and _name_, a varchar field. The table below shows the data we will use for demonstration purposes.

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Foo</td>
</tr>
<tr>
<td>2</td>
<td>Bar</td>
</tr>
</tbody>
</table>

Which methods we choose to use is dependent on three things: the data we want, the format in which we want it, and our personal preference. Much of the Joomla! core prefers methods that return objects.

For the purpose of these examples we won't bother using the _nameQuote()_ and _Quote()_ methods.
The Database

**loadResult( ) : string**

This method loads value of the first cell in the result set. If we selected all the data from our table, this method would return the ID for the first record, in this example: 1. This is useful when we want to access a single field in a known record. For example, we might want to know the name of record 2:

```php
$query = 'SELECT 'name' FROM '#__test' WHERE 'id'=2';
$db =& JFactory::getDBO();
$db->setQuery($query);
echo $db->loadResult();
```

Bar

**loadResultArray( numinarray : int=0 ) : array**

This method loads a column. numinarray is used to specify which column to get; the column is identified by its logical position in the result set.

```php
$query = 'SELECT 'name' FROM '#__test'';
$db =& JFactory::getDBO();
$db->setQuery($query);
print_r($db->loadResultArray());
```

Array

```
[0] => Foo
[1] => Bar
```

**loadAssoc( ) : array**

This method loads the first record as an associative array using the table column names as array keys. This is useful when we are only dealing with an individual record. If the query returns more than one record, the first record in the result set will be used:

```php
$query = 'SELECT * FROM '#__test'';
$db =& JFactory::getDBO();
$db->setQuery($query);
print_r($db->loadAssoc());
```

Array

```
[id] => 1
[name] => Foo
```
loadAssocList( key : string='' ) : array

This method loads an array of associative arrays or an associative array of associative arrays. If we specify the parameter key, the returned array uses the record key as the array key:

```php
$query = 'SELECT * FROM #__test';
$db =& JFactory::getDBO();
$db->setQuery($query);
print_r($db->loadAssocList());
```

Array
(  
[0] => Array  
(  
[id] => 1  
[name] => Foo  
)  
[1] => Array  
(  
[id] => 2  
[name] => Bar  
)  
)

loadObject( ) : stdClass

This method loads the first record as an object using the table column names as property names. This is useful when we are only dealing with an individual record. If the query returns more than one record, the first record in the result set will be used:

```php
$query = 'SELECT * FROM #__test';
$db =& JFactory::getDBO();
$db->setQuery($query);
print_r($db->loadObject());
```

stdClass Object
(  
[id] => 1  
[name] => Foo  
)
The Database

**loadObjectList( key : string='' ) : array**

This method loads an array of stdClass objects or an associative array of stdClass objects. If we specify the parameter `key`, the returned array uses the record key as the array key:

```php
$query = 'SELECT * FROM #__test';
$db =& JFactory::getDBO();
$db->setQuery($query);
print_r($db->loadObjectList());
```

Array

(  
[0] =&gt; stdClass Object
  (  
    [id] =&gt; 1
    [name] =&gt; Foo
  )

[1] =&gt; stdClass Object
  (  
    [id] =&gt; 2
    [name] =&gt; Bar
  )
)

**loadRow( ) : array**

This method loads the first record as an array. This is useful when we are only dealing with an individual record. If the query returns more than one record, the first record in the result set will be used:

```php
$query = 'SELECT * FROM #__test';
$db =& JFactory::getDBO();
$db->setQuery($query);
print_r($db->loadRow());
```

Array

(  
[0] =&gt; 1
[1] =&gt; Foo
)
loadRowList( key : int ) : array

This method loads an array of arrays or an associative array of arrays. If we specify the parameter key, the returned array uses the record key as the array key. Unlike the other load list methods, key is the logical position of the primary key field in the result set:

```php
$query = 'SELECT * FROM #__test';
$db =& JFactory::getDBO();
$db->setQuery($query);
print_r($db->loadRowList(0));
Array
{
    [0] => Array
    {
        [0] => 1
        [1] => Foo
    }
    [1] => Array
    {
        [0] => 2
        [1] => Bar
    }
}
```

Using ADOdb

ADOdb is a PHP database abstraction layer released under the BSD license. ADOdb supports a number of leading database applications. Joomla! does not use ADOdb, but it does emulate some ADOdb functionality in its own database abstraction layer.

We should only use the ADOdb methods if we are porting existing applications that rely on ADOdb or if we are creating extensions that we also want to work as standalone applications using ADOdb.

Joomla! uses the JRecordSet class to emulate the ADOdb ADORecordSet class. The JRecordSet class is not yet complete and does not include all of the ADORecordSet methods. This example shows the basic usage of JRecordSet; $row is an array:

```php
$db =& JFactory::getDBO();
$rs = $db->Execute('SELECT * FROM #__test');
while ($row = $rs->FetchRow())
{
    // process $row
}
```
The Database

For more information about ADOdb, go to http://adodb.sourceforge.net/.

Although ADOdb emulation is being added to Joomla!, it should be noted that there are currently no plans to integrate ADOdb as the primary means of accessing the Joomla! database.

JTable

Joomla! provides us with the powerful abstract class JTable; with this we can perform many basic functions on table records. For every table that we want to use the JTable class with, we must create a new subclass.

When creating JTable subclasses we must follow some specific conventions. These conventions enable us to integrate our extensions into Joomla! and the Joomla! framework.

Assuming we are building a component, our JTable subclasses should be located in separate files in a folder called tables within the component's administrative root. The class name is the table singular entity name prefixed with Table. The name of the file is the singular entity name.

We will use the table schema, which we defined earlier in this chapter, for the entity foobar in the extension 'My Extension', to demonstrate how we use JTable in conjunction with a database table. You may want to familiarize yourself with the schema before continuing.

The class is called TableFoobar and is located in the file JPATH_COMPONENT_ADMINISTRATOR.DS.'tables'.DS.'foobar.php'. The first thing we need to do in our class is to define the public properties. The public properties relate directly to the fields and must have exactly the same names. We use these properties as a 'buffer' to store individual records.

The second thing we need to do is to define the constructor. In order to use the JTable::getInstance() method, we must override the JTable constructor with a constructor that has a single referenced parameter, the database object.

The third thing we need to do is override the check() method. This method is used to validate the buffer contents, returning a Boolean result. If a check() fails we use the setError() method to set a message that explains the reason why the validation failed.
/**
 * #__myextension_foobars table handler
 * /
class TableFoobar extends JTable
{
    /** @var int Primary key */
    var $id = null;
    /** @var string Content */
    var $content = null;
    /** @var int Checked-out owner */
    var $checked_out = null;
    /** @var string Checked-out time */
    var $checked_out_time = null;
    /** @var string Parameters */
    var $params = null;
    /** @var int Order position */
    var $ordering = null;
    /** @var int Number of views */
    var $hits = null;

    /**
     * Constructor
     *
     * @param database Database object
     */
    function __construct( &$db )
    {
        parent::__construct('#__myextension_foobars', 'id', $db);
    }

    /**
     * Validation
     *
     * @return boolean True if buffer is valid
     */
    function check()
    {
        if(!$this->content)
        {
            $this->setError(JText::_('Your Foobar must contain some content'));
            return false;
        }
        return true;
    }
}
The Database

Now that we have created our TableFoobar class what do we do with it? Well first of all we need to instantiate a TableFoobar object using the static JTable::getInstance() method.

JTable::addIncludePath(JPATH_COMPONENT_ADMINISTRATOR.DS.'tables');
$table = JTable::getInstance('foobar', 'Table');

Note that instead of including the foobar.php file, we tell JTable where the containing folder is. When JTable comes to instantiate the TableFoobar object, if the class is not defined, it will look in all of the JTable include paths for a file named foobar.php.

**CRUD**

**CRUD (Create Read Update Delete)** is the name given to the four common data manipulation tasks. We will follow a record through its short 'CRUDy' life. Throughout the CRUD examples $table refers to an instance of the TableFoobar class and $id refers to the ID of the record we are dealing with. In this example, we create a new record; $table is an instance of the TableFoobar class.

```php
$table->reset();
$table->set('content', "Lorem ipsum dolor sit amet");
$table->set('ordering', $table->getNextOrder());
if ($table->check())
{
    if (!$table->store())
    {
        // handle failed store
        // use $table->getError() for an explanation
    }
}
else
{
    // handle failed check
    // use $table->getError() for an explanation
}
```

The reset() method ensures that the table buffer is empty. The method returns all of the properties to their default values specified by the class. The getNextOrder() method determines the next space in the record ordering. If there are no existing records, this will be 1. In case the check() method returns false, we should have some handling in place. In most circumstances using a redirect and en-queuing the check() error message will suffice.
Let us tidy up our example. Some of the fields have default values defined in the table, so our buffer will not be up to date after the record is created. When we create a new record because the class knows what the table primary key is, the primary key buffer property is automatically updated. After the previous example the buffer for $table looks like this:

```
[id] => 1
[content] => Lorem ipsum dolor sit amet
[checked_out] =>
[checked_out_time] =>
[params] =>
[ordering] => 1
[hits] => 0
```

After storing the new record, we can load the record from the database ensuring that the buffer is up to date. This example loads the new record from the table into the buffer.

```
$table->load($table->id);
```

Now the buffer will look like this:

```
[id] => 1
[content] => Lorem ipsum dolor sit amet
[checked_out] => 0
[checked_out_time] => 0000-00-00 00:00:00
[params] =>
[ordering] => 1
[hits] => 0
```

Instead of loading newly added records, we could modify the TableFoobar class so that the default values correspond directly to the database table's default values. This way we reduce our overheads and do not have to reload the record.

However, because some of the default values are dependent upon the database, to do this we would have to modify the constructor and override the `reset()` method. For example the `checked_out_time` field default value is `$db->getNullDate()`, and we cannot use this when defining parameters.
The way we updated the table buffer after creating the new record is precisely the same way we would load (read) any existing record. This example shows how we load a record into the buffer:

```php
if (!$table->load($id))
{
    // handle unable to load
    // use $table->getError() for an explanation
}
```

Well, we are steaming through this CRUD (not literally). Next up is updating an existing record. There are two ways of updating a record. We can insert the updated data into the buffer and update the record. Alternatively, we can load the record, insert the updated data into the buffer, and update the record. This example shows how we implement the simpler first option:

```php
// set values
$table->reset();
$table->setVar('id', $id);
$table->setVar('content', JRequest::getString('content'));
if ($table->check())
{
    if (!$table->store())
    {
        // handle failed update
        // use $table->getError() for an explanation
    }
}
else
{
    // handle invalid input
    // use $table->getError() for an explanation
}
```

Although this works, if it fails, we do not even know whether it is due to an invalid record ID or a more complex problem. There is a quirk we need to be aware of when using the `store()` method. It only updates the values that are not null; we can force it to update nulls, by passing a `true` parameter to the `store` method. The issue with this is we would need to have the record loaded into the buffer so that we do not overwrite anything with null values. This example demonstrates how we can implement this.

```php
if ($table->load($id))
{
    // handle failed load
    // use $table->getError() for an explanation
}
```
else
{
	$table->setVar('content', JRequest::getString('content'));
	if ($table->check())
	{
		if (!$table->store(true))
		{
			// handle failed update
			// use $table->getError() for an explanation
		}
	}
	else
	{
		// handle invalid input
		// use $table->getError() for an explanation
	}
}

The last action that will occur in any record's life is deletion. Deleting a record using JTable subclasses is very easy. This example shows how we delete a record.

if (!$table->delete($id))
{
	// handle failed delete
}

If we don't pass an ID to the delete() method, the ID in the buffer will be used. It is important to bear in mind that if you do pass an ID the buffer ID will be updated.

If we are deleting a record that has relationships with other tables, we can check for dependencies using the canDelete() method. The canDelete() method has one parameter, a two dimensional array. The inner arrays must contain the keys, idfield, name, joinfield, and label. idfield is the name of the primary key in the related table. name is the name of the related table. joinfield is the name of the foreign key in the related table. label is the description of the relationship to use in the error message if any dependencies are found.

Imagine that there is another table called #__myextension_children; this table has a primary key called childid and a foreign key called parent, which is related to the primary key field id in #__myextension_foobars. In this example, we verify there are no dependent records in the #__myextension_children table before deleting a record from #__myextension_foobars.

$join1 = array('idfield' => 'childid',
	'name' => '__myextension_children',
	'joinfield' => 'parent',
	'label' => 'Children');
$joins = array($join1);
if ($table->canDelete($id, $joins))
{
    if (!$table->delete($id))
    {
        // handle failed delete
        // use $table->getError() for an explanation
    }
}
else
{
    // handle dependent records, cannot delete
    // use $table->getError() for an explanation
}

We can define more than one join, for example had there been another table called #__myextension_illegitimate_children we could also have defined this in the $joins array.

$join1 = array('idfield' => 'childid',
               'name' => '#__myextension_children',
               'joinfield' => 'parent',
               'label' => 'Children');
$join2 = array('idfield' => 'ichildid',
               'name' => '#__myextension_illegitimate_children',
               'joinfield' => 'parent',
               'label' => 'illegitimate Children');
$joins = array($join1, $join2);

The names of primary keys and foreign keys in all of the tables must not be the same as the names of any other fields in any of the other tables. Otherwise, the query will become ambiguous and the method will always return false.

### Manipulating Common Fields

Let us rewind a bit, killing off our record in its prime was a little mean after all! Our table includes all of those handy common fields we mentioned earlier and JTable provides us with some useful methods for dealing specifically with those fields. Throughout the Common Fields examples $table refers to an instance of the TableFoobar class and $id refers to the ID of the record we are dealing with.
Publishing

To publish and un-publish data we can use the publish() method. This method publishes and un-publishes multiple records at once. If the table includes a checked_out field, we can ensure that the record is not checked out or is checked out to the current user. This example publishes a record.

```php
$publishIds = array($id);
$user =& JFactory::getUser();
if (!$table->publish($publishIds, 1, $user->get('id')))
{
    // handle unable to publish record
    // use $table->getError() for an explanation
}
```

The first parameter is an array of keys of the records we wish to publish or un-publish. The second parameter is the new published value, 0 = not published, 1 = published; this is optional, by default it is 1. The final parameter, also optional, is used only when the checked_out field exists. Only fields that are not checked out or are checked out by the specified user can be updated.

The method returns true if the publishing was successful. This is not the same as saying all the specified records have been updated. For example if a specified record is checked out by a different user, the record will not be updated but the method will return true.

Hits

To increment the hits field we can use the hit() method. In this example we set the buffer record ID and use the hit() method.

```php
$table->set('id', $id);
$table->hit();
```

Alternatively we can specify the ID when we use the hit() method. If we choose to do this, we must remember that the buffer ID will be updated to match the hit ID.

```php
$table->hit($id);
```

Checking Out

Before we start checking out records, we first need to check if a record is already checked out. Remember that when a record is checked out we should not allow any other user to modify the record. We can use the isCheckOutMethod() to achieve this. In this example, we test to see if any user, other than the current user, has checked out the record:
$table->load($id);
$user =& JFactory::getUser();
if ($table->isCheckedOut($user->get('id')))
{
    // handle record is already checked-out
}

Once we have determined a record isn't checked out, we can use the `checkout()` method to check out the record. In this example, we check out the record to the current user; this sets the `checked_out` field to the user's ID and the `checked_out_time` field to the current time.

$table->load($id);
$user =& JFactory::getUser();
if (!$table->checkout($user->get('id')))
{
    // handle failed to checkout record
}

Now that we have a checked-out record, we need to know how to check it in. To do this we use the `checkin()` method. This example checks in a record; this will set the `checked_out_time` field to a null date:

$table->load($id);
$user =& JFactory::getUser();
if (!$table->checkin($user->get('id')))
{
    // handle failed to checkin record
}

We should only check records in and out for logged in users. For a more comprehensive check-out system use Joomla!'s access control system explained in Chapter 11.

Ordering

When we want to order items, JTable gives us a number of useful methods. The first one of these we will look at is `reorder()`. This method looks at each record and moves them up the order chain until any gaps in the order have been removed. In this example, we reorder our table:

$table->reorder();

Very simple, but for more complicated tables there could be groupings within the records. To deal with this we can provide the `reorder()` method with a parameter to restrict the records. Imagine that our table also has a field named `group`; in this case...
example, we reorder the records in group 1:

```php
$db =& $table::getDBO();
$where = $db->nameQuote('group').' = 1';
$table->reorder($where);
```

Notice that we get the database object from $table not JFactory; this ensures that we are using the correct database driver for the database server that $table is using. Although this is not a major issue, as Joomla! begins to support other database drivers, there may be occasions where the database driver being used by a table is different from the global database driver.

You may remember earlier in this chapter we used the `getNextOrder()` method. This method tells us what the next available position is in the order. As with `reorder()`, we have the option of specifying groupings. Imagine that our table also has a field named `group`; in this example, we get the next available position in the records in group 1:

```php
$db =& $table::getDBO();
$where = $db->nameQuote('group').' = 1';
$nextPosition = $table->getNextOrder($where);
```

Last of all we can use the `move()` method to move a record up or down one position. In this example, we move a record up the order:

```php
$table->load($id);
$table->move(-1);
```

Again, we have the option of specifying groupings. Imagine that our table also has a field named `group`; in this example, we move a record down the order in group 1:

```php
$db =& $table::getDBO();
$where = $db->nameQuote('group').' = 1';
$table->load($id);
$table->move(1, $where);
```

**Parameter Fields**

The JTable class does not provide us with any special methods for dealing with INI parameter fields. The JTable buffer is designed to be populated with the RAW data, as it will be stored in the database.

To handle a parameter field we use the JParameter class. The first thing we need to do is create a new JParameter object and, if we are interrogating an existing record, parse the parameter data.
The Database

The JParameter class extends the JRegistry class; the JRegistry class is explained in Chapter 7. This example shows how we can parse INI data using the JParameter class:

```php
$params = new JParameter($table->params);
```

Once we have a JParameter object we can access and modify the data in the object using the `get()` and `set()` methods:

```php
$value = $params->get('someValue');
$params->set('someValue', ++$value);
```

We can return the data to an INI string using the `toString()` method:

```php
$table->params = $params->toString();
```

We can also use the JParameter class in conjunction with an XML metadata file to define the values we might be holding in an INI string. This example shows how we create a new JParameter object and load an XML metadata file; `$path` is the full path to an XML manifest file:

```php
$params = new JParameter('foo=bar', $pathToXML_File);
```

There is a full description explaining how to define an XML metadata file for these purposes in Chapter 4 and the Appendix. We can use the `render()` method to output form elements populated with the parameter values (how these are rendered is defined in the XML file):

```php
echo $params->render('params');
```

Date Fields

Different database servers use different date and time formats to store dates and times. It is important that when we come to save dates and times we use the correct format for the database that is being used.

Sadly, there is currently no way to ensure that we are using the format specific to the database being used. Instead we must assume that the database is MySQL based. This means that we must store dates in the format `YYYY-MM-DD HH:MM:SS`.

The easiest way to do this is to use the JDate class. JDate objects are used to parse and represent date and time values. We use the `toMySQL()` method to ensure that the value is formatted appropriately:

```php
// import JDate class
jimport('joomla.utilities.date');

// get current date and time (unix timestamp)
$myDate = gmdate();
```
// create JDate object
$jdate = new JDate($myDate);

// create query using toMySQL()
$query = 'SELECT * FROM #__example WHERE date < '.$jdate->toMySQL();

The value that we pass when creating the JDate object can be in the format UNIX timestamp, RFC 2822 / 822, or ISO 8601. A more complete description of JDate is available in Chapter 12.

Summary
We should now be able to successfully create new database table schemas; how we add these tables to the database is explained in more detail in the next chapter, Chapter 4. We can build queries that are ready for use with our specific database driver using the nameQuote() and Quote() methods. We must remember to use these two methods; if we do not we run the risk of restricting our queries to MySQL databases.

We can extend the abstract JTable class adding an extra element to the data access layer. JTable allows us to perform many common actions on records. Taking advantage of the JTable class can significantly reduce the overheads incurred while programming and it ensures that we use standardized methods to perform actions.
Where to buy this book


Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.