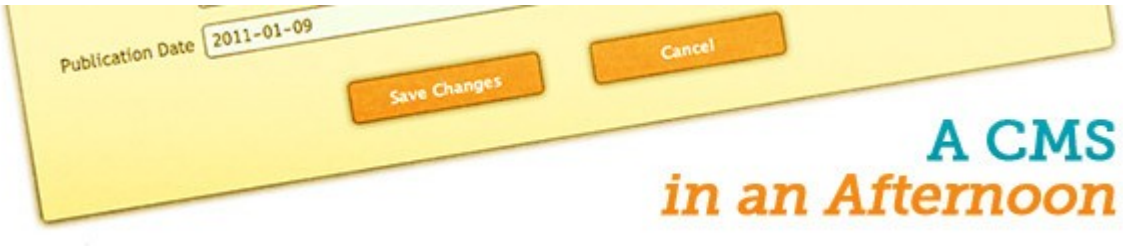
Build a CMS in an Afternoon with PHP and MySQL

/ 600 Comments



View Demo » | **Download Code**

14 May 2019: This article and the code were updated for PHP7 compatibility.

Building a content management system can seem like a daunting task to the novice PHP developer. However, it needn't be that difficult. In this tutorial I'll show you how to build a basic, but fully functional, CMS from scratch in just a few hours. Yes, it can be done!

Along the way, you'll learn how to create MySQL databases and tables; how to work with PHP objects, constants, includes, sessions, and other features; how to separate business logic from presentation; how to make your PHP code more secure, and much more!

Before you begin, check out the finished product by clicking the *View Demo* link above. (For security reasons this demo is read-only, so you can't add, change or delete articles.) You can also click the *Download Code* link above to download the complete PHP code for the CMS, so you can run it on your own server.

XAMPP on your computer.

The feature list

Our first job is to work out exactly what we want our CMS to do. The CMS will have the following features:

Front end:

- The homepage, listing the 5 most recent articles
- The article listing page, listing all articles
- The "view article" page, letting visitors see a single article

Back end:

- Admin login/logout
- List all articles
- Add a new article
- Edit an existing article

- Delete an existing article

Each article will have an associated headline, summary, and publication date.

Planning it out

Here are the steps we'll need to follow to create our CMS:

1. Create the database
2. Create the `articles` database table
3. Make a configuration file
4. Build the `Article` class
5. Write the front-end `index.php` script
6. Write the back-end `admin.php` script
7. Create the front-end templates
8. Create the back-end templates
9. Create the stylesheet and logo image

finished zip file, which contains all the code files and folders.

Ready? Grab a cup of tea, and let's get coding!

Step 1: Create the database

The first thing we need to do is create a MySQL database to store our content. You can do this as follows:

1. **Run the `mysql` client program**
   Open a terminal window and enter the following:

   ```
   mysql -u username -p
   ```

   Then enter your MySQL password when prompted.

   `username` should be a user that has permission to create databases. If you're working on a development server, such as your own computer, then you can use the `root` user for this, to save having to create a new user.
2. **Create the database**
   At the `mysql>` prompt, type:

```
create database cms;
```

Then press Enter.

3. **Quit the `mysql` client program**
   At the `mysql>` prompt, type:

   ```
   exit
   ```

   Then press Enter.

That's it! You've now created a new, empty database, into which you can put your database tables and content.

Step 2: Create the `articles` database table

Our simple CMS has just one database table: `articles`. This, as you'd imagine, holds all of the articles in the system.

Let's create the schema for the table. A table's schema describes the types of data that the table can hold, as well as other information about the table.

Create a text file called `tables.sql` somewhere on your hard drive. Add the following code to the file:

```
DROP TABLE IF EXISTS articles;
CREATE TABLE articles
(
  id              smallint unsigned NOT NULL auto_increment,
  publicationDate date NOT NULL,                              # When the article
was published
  title           varchar(255) NOT NULL,                      # Full title of
the article
  summary         text NOT NULL,                              # A short summary
of the article
  content         mediumtext NOT NULL,                        # The HTML content
of the article

  PRIMARY KEY     (id)
);
```

The above code defines the schema for the `articles` table. It's written in SQL, the language used to create and manipulate databases in MySQL (and most other database systems).

Let's break the above code down a little:

1. **Create the `articles` table**

   `DROP TABLE IF EXISTS articles` removes any existing `articles` table (and data — be careful!) if it already exists. We do this because we can't define a table with the same name as an existing table.

   `CREATE TABLE articles ( )` creates the new `articles` table. The stuff inside the parentheses defines the structure of the data within the table, explained below…

2. **Give each article a unique ID**

We're now ready to define our table structure. A table consists of a number of fields (also called columns). Each field holds a specific type of information about each article.

First, we create an `id` field. This has a `smallint unsigned` (unsigned small integer) data type, which means it can hold whole numbers from 0 to 65,535. This lets our CMS hold up to 65,535 articles. We also specify the `NOT NULL` attribute, which means the field can't be empty (null) — this makes life easier for us. We also add the `auto_increment` attribute, which tells MySQL to assign a new, unique value to an article's `id` field when the article record is created. So the first article will have an `id` of 1, the second will have an `id` of 2, and so on. We'll use this unique value as a handle to refer to the article that we want to display or edit in the CMS.

3. **Add the `publicationDate` field**
   The next line creates the `publicationDate` field, which stores the date that each article was published. This field has a data type of `date`, which means it can store date values.
4. **Add the `title` field**
   Next we create the `title` field to hold each article's title. It has a data type of `varchar(255)`, which means it can store a string of up to 255 characters.
5. **Add the `summary` and `content` fields**
   The last 2 fields, `summary` and `content`, hold a short summary of the article and the article's HTML content respectively. summary has a `text` data type (which can hold up to 65,535 characters) and `content` has a `mediumtext` data type (which can hold up to 16,777,215 characters).
6. **Add the primary key**

   The last line inside the `CREATE TABLE` statement defines a key for the table. A key is also called an index, and in simple terms it makes it quicker to find data in the table, at the expense of some extra storage space.

   We make the `id` field a `PRIMARY KEY`. Each table can only have a single `PRIMARY KEY`; this is the key that uniquely identifies each record in the table. In addition, by adding this key, MySQL can retrieve an article based on its ID very quickly.

Now that we've created our table schema, we need to load it into MySQL to create the table itself. The easiest way to do this is to open up a terminal window and change to the folder containing your `tables.sql` file, then run this command:

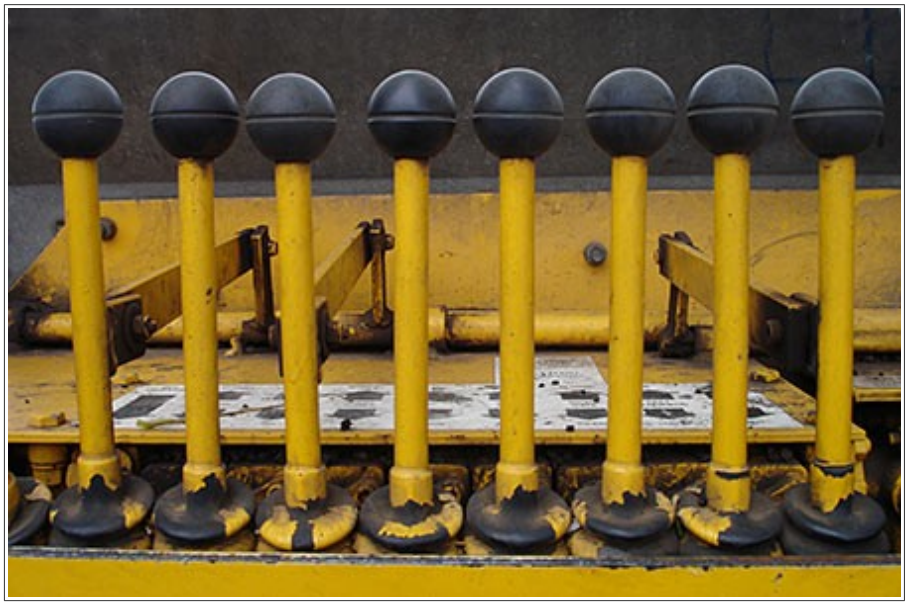
```
mysql -u username -p cms < tables.sql
```

…where *`username`* is your MySQL username. `cms` is the name of the database that you created in Step 1.

Enter your password when prompted. MySQL then loads and runs the code in your `tables.sql` file, creating the `articles` table inside the `cms` database.

phpMyAdmin to run your `tables.sql` code and create the table. phpMyAdmin comes pre-installed with most web hosting accounts.

Step 3: Make a configuration file



Now that you've created your database, you're ready to start writing your PHP code. Let's start by creating a configuration file to store various useful settings for our CMS. This file will be used by all the script files in our CMS.

First, create a `cms` folder somewhere in the local website on your computer, to hold all the files relating to the CMS. If you're running XAMPP then the local website will be in an `htdocs` folder inside your XAMPP folder. Or, if you prefer, you can create a brand new website just for your CMS, and put all the files in that new website's document root folder.

Inside the `cms` folder, create a file called `config.php` with the following code:

```
<?php
ini_set( "display_errors", true );
date_default_timezone_set( "Australia/Sydney" );  //
http://www.php.net/manual/en/timezones.php
define( "DB_DSN", "mysql:host=localhost;dbname=cms" );
define( "DB_USERNAME", "username" );
define( "DB_PASSWORD", "password" );
define( "CLASS_PATH", "classes" );
define( "TEMPLATE_PATH", "templates" );
define( "HOMEPAGE_NUM_ARTICLES", 5 );
define( "ADMIN_USERNAME", "admin" );
define( "ADMIN_PASSWORD", "mypass" );
require( CLASS_PATH . "/Article.php" );

function handleException( $exception ) {
  echo "Sorry, a problem occurred. Please try later.";
  error_log( $exception->getMessage() );
}

set_exception_handler( 'handleException' );
?>
```

Let's break this file down:

1. **Display errors in the browser**
   The `ini_set()` line causes error messages to be displayed in the browser. This is good for debugging, but it should be set to `false` on a live site since it can be a security risk.
2. **Set the timezone**
   As our CMS will use PHP's `date()` function, we need to tell PHP our server's timezone (otherwise PHP generates a warning message). Mine is set to `"Australia/Sydney"` — change this value to your local timezone.
3. **Set the database access details**
   Next we define a constant, `DB_DSN`, that tells PHP where to find our MySQL database. Make sure the `dbname` parameter matches the name of your CMS database (`cms` in this case). We also store the MySQL username and password that are used to access the CMS database in the constants `DB_USERNAME` and `DB_PASSWORD`. Set these values to your MySQL username and password.
4. **Set the paths**
   We set 2 path names in our config file: `CLASS_PATH`, which is the path to the class files, and `TEMPLATE_PATH`, which is where our script should look for the HTML template files. Both these paths are relative to our top-level `cms` folder.
5. **Set the number of articles to display on the homepage**
   `HOMEPAGE_NUM_ARTICLES` controls the maximum number of article headlines to display on the site homepage. We've set this to 5 initially, but if you want more or less articles, just change this value.
6. **Set the admin username and password**
   The `ADMIN_USERNAME` and `ADMIN_PASSWORD` constants contain the login details for the CMS admin user. Again, you'll want to change these to your own values.
7. **Include the `Article` class**
   Since the `Article` class file — which we'll create next — is needed by all scripts in our application, we include it here.
8. **Create an exception handler**
   Finally, we define `handleException()`, a simple function to handle any PHP exceptions that might be raised as our code runs. The function displays a generic error message, and logs the actual exception message to the web server's error log. In particular, this function improves security by handling any PDO exceptions that might otherwise display the database username and password in the page. Once we've defined `handleException()`, we set it as the exception handler by calling PHP's `set_exception_handler()` function. `Article.php` in `try ... catch` blocks.

Security note

In a live server environment it'd be a good idea to place `config.php` somewhere outside your website's document root, since it contains usernames and passwords. While it's not usually possible to read the source code of a PHP script via the browser, it does happen sometimes if the web server is misconfigured.

You could also use `hash()` to make a hash from your admin password, and store the hash in `config.php` instead of the plaintext password. Then, at login time, you can `hash()` the entered password and see if it matches the hash in `config.php`.

Step 4: Build the `Article` class

You're now ready to build the `Article` PHP class. This is the only class in our CMS, and it handles the nitty-gritty of storing articles in the database, as well as retrieving articles from the database. Once we've built this class, it will be really easy for our other CMS scripts to create, update, retrieve and delete articles.

Inside your `cms` folder, create a `classes` folder. Inside that `classes` folder, create a new file called `Article.php`, and put the following code into it:

```
<?php

/**
 * Class to handle articles
 */

class Article
{

  // Properties

  /**
  * @var int The article ID from the database
  */
  public $id = null;

  /**
  * @var int When the article was published
  */
  public $publicationDate = null;
```

```
  /**
  * @var string Full title of the article
  */
  public $title = null;

  /**
  * @var string A short summary of the article
  */
  public $summary = null;

  /**
  * @var string The HTML content of the article
  */
  public $content = null;


  /**
  * Sets the object's properties using the values in the supplied array
  *
  * @param assoc The property values
  */

  public function __construct( $data=array() ) {
    if ( isset( $data['id'] ) ) $this->id = (int) $data['id'];
    if ( isset( $data['publicationDate'] ) ) $this->publicationDate = (int)
$data['publicationDate'];
    if ( isset( $data['title'] ) ) $this->title = preg_replace ( "/[^\.\,\-\
_\'\"\@\?\!\:\$ a-zA-Z0-9()]/", "", $data['title'] );
    if ( isset( $data['summary'] ) ) $this->summary = preg_replace ( "/[^\.\,\-\
_\'\"\@\?\!\:\$ a-zA-Z0-9()]/", "", $data['summary'] );
    if ( isset( $data['content'] ) ) $this->content = $data['content'];
  }


  /**
  * Sets the object's properties using the edit form post values in the supplied
array
  *
  * @param assoc The form post values
  */

  public function storeFormValues ( $params ) {

    // Store all the parameters
    $this->__construct( $params );

    // Parse and store the publication date
    if ( isset($params['publicationDate']) ) {
      $publicationDate = explode ( '-', $params['publicationDate'] );

      if ( count($publicationDate) == 3 ) {
        list ( $y, $m, $d ) = $publicationDate;
        $this->publicationDate = mktime ( 0, 0, 0, $m, $d, $y );
      }
    }
  }


  /**
  * Returns an Article object matching the given article ID
  *
  * @param int The article ID
  * @return Article|false The article object, or false if the record was not
found or there was a problem
```

```
  */

  public static function getById( $id ) {
    $conn = new PDO( DB_DSN, DB_USERNAME, DB_PASSWORD );
    $sql = "SELECT *, UNIX_TIMESTAMP(publicationDate) AS publicationDate FROM
articles WHERE id = :id";
    $st = $conn->prepare( $sql );
    $st->bindValue( ":id", $id, PDO::PARAM_INT );
    $st->execute();
    $row = $st->fetch();
    $conn = null;
    if ( $row ) return new Article( $row );
  }


  /**
  * Returns all (or a range of) Article objects in the DB
  *
  * @param int Optional The number of rows to return (default=all)
  * @return Array|false A two-element array : results => array, a list of
Article objects; totalRows => Total number of articles
  */

  public static function getList( $numRows=1000000 ) {
    $conn = new PDO( DB_DSN, DB_USERNAME, DB_PASSWORD );
    $sql = "SELECT SQL_CALC_FOUND_ROWS *, UNIX_TIMESTAMP(publicationDate) AS
publicationDate FROM articles
            ORDER BY publicationDate DESC LIMIT :numRows";

    $st = $conn->prepare( $sql );
    $st->bindValue( ":numRows", $numRows, PDO::PARAM_INT );
    $st->execute();
    $list = array();

    while ( $row = $st->fetch() ) {
      $article = new Article( $row );
      $list[] = $article;
    }

    // Now get the total number of articles that matched the criteria
    $sql = "SELECT FOUND_ROWS() AS totalRows";
    $totalRows = $conn->query( $sql )->fetch();
    $conn = null;
    return ( array ( "results" => $list, "totalRows" => $totalRows[0] ) );
  }


  /**
  * Inserts the current Article object into the database, and sets its ID
property.
  */

  public function insert() {

    // Does the Article object already have an ID?
    if ( !is_null( $this->id ) ) trigger_error ( "Article::insert(): Attempt to
insert an Article object that already has its ID property set (to $this->id).",
E_USER_ERROR );

    // Insert the Article
    $conn = new PDO( DB_DSN, DB_USERNAME, DB_PASSWORD );
    $sql = "INSERT INTO articles ( publicationDate, title, summary, content )
VALUES ( FROM_UNIXTIME(:publicationDate), :title, :summary, :content )";
    $st = $conn->prepare ( $sql );
```

```
    $st->bindValue( ":publicationDate", $this->publicationDate,
PDO::PARAM_INT );
    $st->bindValue( ":title", $this->title, PDO::PARAM_STR );
    $st->bindValue( ":summary", $this->summary, PDO::PARAM_STR );
    $st->bindValue( ":content", $this->content, PDO::PARAM_STR );
    $st->execute();
    $this->id = $conn->lastInsertId();
    $conn = null;
  }


  /**
  * Updates the current Article object in the database.
  */

  public function update() {

    // Does the Article object have an ID?
    if ( is_null( $this->id ) ) trigger_error ( "Article::update(): Attempt to
update an Article object that does not have its ID property set.",
E_USER_ERROR );

    // Update the Article
    $conn = new PDO( DB_DSN, DB_USERNAME, DB_PASSWORD );
    $sql = "UPDATE articles SET publicationDate=FROM_UNIXTIME(:publicationDate),
title=:title, summary=:summary, content=:content WHERE id = :id";
    $st = $conn->prepare ( $sql );
    $st->bindValue( ":publicationDate", $this->publicationDate,
PDO::PARAM_INT );
    $st->bindValue( ":title", $this->title, PDO::PARAM_STR );
    $st->bindValue( ":summary", $this->summary, PDO::PARAM_STR );
    $st->bindValue( ":content", $this->content, PDO::PARAM_STR );
    $st->bindValue( ":id", $this->id, PDO::PARAM_INT );
    $st->execute();
    $conn = null;
  }


  /**
  * Deletes the current Article object from the database.
  */

  public function delete() {

    // Does the Article object have an ID?
    if ( is_null( $this->id ) ) trigger_error ( "Article::delete(): Attempt to
delete an Article object that does not have its ID property set.",
E_USER_ERROR );

    // Delete the Article
    $conn = new PDO( DB_DSN, DB_USERNAME, DB_PASSWORD );
    $st = $conn->prepare ( "DELETE FROM articles WHERE id = :id LIMIT 1" );
    $st->bindValue( ":id", $this->id, PDO::PARAM_INT );
    $st->execute();
    $conn = null;
  }

}

?>
```

This file is quite long, but it’s fairly simple stuff when you break it down. Let’s take a look at each section of the code:

1. The class definition and properties

First, we begin to define our `Article` class with the code:

```
class Article
{
```

Everything after these lines of code — up until the closing brace at the end of the file — contains the code that makes up the `Article` class.

After starting our class definition, we declare the properties of the class: `$id`, `$publicationDate`, and so on. Each `Article` object that we create will store its article data in these properties. You can see that the property names mirror the field names in our `articles` database table.

active record.

2. The constructor

Next we create the class methods. These are functions that are tied to the class, as well as to objects created from the class. Our main code can call these methods in order to manipulate the data in the `Article` objects.

The first method, `__construct()`, is the constructor. This is a special method that is called automatically by the PHP engine whenever a new `Article` object is created. Our constructor takes an optional `$data` array containing the data to put into the new object's properties. We then populate those properties within the body of the constructor. This gives us a handy way to create and populate an object in one go.

`$this->`*`propertyName`* means: "The property of this object that has the name "*`$propertyName`*".

You'll notice that the method filters the data before it stores them in the properties. The `id` and `publicationDate` properties are cast to integers using `(int)`, since these values should always be integers. The `title` and `summary` are filtered using a regular expression to only allow a certain range of characters. It's good security practice to filter data on input like this, only allowing acceptable values and characters through.

We don't filter the `content` property, however. Why? Well, the administrator will probably want to use a wide range of characters, as well as HTML markup, in the article content. If we restricted the range of allowed characters in the content then we would limit the usefulness of the CMS for the administrator.

Normally this could be a security loophole, since a user could insert malicious JavaScript and other nasty stuff in the content. However, since we presumably trust our site administrator — who is the only person allowed to create the content — this is an acceptable tradeoff in this case. If you were dealing with user-generated content, such as comments or forum posts, then you would want to be more careful, and only allow "safe" HTML to be used. A really great tool for this is HTML Purifier, which thoroughly analyses HTML input and removes all potentially malicious code.

Filter Input-Escape Output: Security Principle and Practice. Also see the Wikipedia entries on secure input/output handling, XSS, CSRF, SQL injection, and session fixation.

3. `storeFormValues()`

Our next method, `storeFormValues()`, is similar to the constructor in that it stores a supplied array of data in the object's properties. The main difference is that `storeFormValues()` can handle data in the format that is submitted via our New Article and Edit Article forms (which we'll create later). In particular, it can handle publication dates in the format `YYYY-MM-DD`, converting the date into the UNIX timestamp format suitable for storing in the object.

The purpose of this method is simply to make it easy for our admin scripts to store the data submitted by the forms. All they have to do is call `storeFormValues()`, passing in the array of form data.

`Article` class have the `public` keyword before their names, which means that they're available to code outside the class. You can also create `private` members (which can only be used by the class itself) and `protected` members (which can be used by the class and any of its subclasses). Don't worry, I'll be covering all this in a later tutorial!

4. `getById()`

Now we come to the methods that actually access the MySQL database. The first of these, `getById()`, accepts an article ID argument (`$id`), then retrieves the article record with that ID from the `articles` table, and stores it in a new `Article` object.

Usually, when you call a method, you first create or retrieve an object, then call the method on that object. However, since this method returns a new `Article` object, it would be helpful if the method could be called directly by our calling code, and not via an existing object. Otherwise, we would have to create a new dummy object each time we wanted to call the method and retrieve an article.

To enable our method to be called without needing an object, we add the `static` keyword to the method definition. This allows the method to be called directly without specifying an object:

```
public static function getById( $id ) {
```

The method itself uses PDO to connect to the database, retrieve the article record using a `SELECT` SQL statement, and store the article data in a new `Article` object, which is then returned to the calling code.

PDO — PHP Data Objects — is an object-oriented library built into PHP that makes it easy for PHP scripts to talk to databases.

Let's break this method down:

1. **Connect to the database**

   ```
   $conn = new PDO( DB_DSN, DB_USERNAME, DB_PASSWORD );
   ```

   This makes a connection to the MySQL database using the login details from the `config.php` file, and stores the resulting connection handle in `$conn`. This handle is used by the remaining code in the method to talk to the database.

2. **Retrieve the article record**

```
    $sql = "SELECT *, UNIX_TIMESTAMP(publicationDate) AS publicationDate
FROM articles WHERE id = :id";
    $st = $conn->prepare( $sql );
    $st->bindValue( ":id", $id, PDO::PARAM_INT );
    $st->execute();
    $row = $st->fetch();
```

Our `SELECT` statement retrieves all fields (`*`) from the record in the `articles` table that matches the given `id` field. It also retrieves the `publicationDate` field in UNIX timestamp format instead of the default MySQL date format, so we can store it easily in our object.

Rather than placing our `$id` parameter directly inside the `SELECT` string, which can be a security risk, we instead use `:id`. This is known as a placeholder. In a minute, we'll call a PDO method to bind our `$id` value to this placeholder.

Once we've stored our `SELECT` statement in a string, we prepare the statement by calling `$conn->prepare()`, storing the resulting statement handle in a `$st` variable.

We now bind the value of our `$id` variable — that is, the ID of the article we want to retrieve — to our `:id` placeholder by calling the `bindValue()` method. We pass in the placeholder name; the value to bind to it; and the value's data type (integer in this case) so that PDO knows how to correctly escape the value.

Lastly, we call `execute()` to run the query, then we use `fetch()` to retrieve the resulting record as an associative array of field names and corresponding field values, which we store in the `$row` variable.

3. **Close the connection**

```
    $conn = null;
```

Since we no longer need our connection, we close it by assigning `null` to the `$conn` variable. It's a good idea to close database connections as soon as possible to free up memory on the server.

4. **Return the new `Article` object**

```
    if ( $row ) return new Article( $row );
  }
```

The last thing our method needs to do is create a new `Article` object that stores the record returned from the database, and return this object to the calling code. First it checks that the returned value from the `fetch()` call, `$row`, does in fact contain data. If it does then it creates a new `Article` object, passing in `$row` as it does so. Remember that this calls our constructor that we created earlier, which populates the object with the data contained in the `$row` array. We then return this new object, and our work here is done.

5. `getList()`

Our next method, `getList()`, is similar in many ways to `getById()`. The main difference, as you might imagine, is that it can retrieve many articles at once, rather than just 1 article. It's used whenever we need to display a list of articles to the user or administrator.

`getList()` accepts an optional argument:

`$numRows`
: The maximum number of articles to retrieve. We default this value to 1,000,000 (i.e. effectively all articles). This parameter allows us to display, say, just the first 5 articles on the site homepage.

Much of this method's code is similar to `getById()`. Let's look at a few lines of interest:

```
    $sql = "SELECT SQL_CALC_FOUND_ROWS *, UNIX_TIMESTAMP(publicationDate) AS
publicationDate FROM articles
            ORDER BY publicationDate DESC LIMIT :numRows";
```

Our query is a bit more complex than last time. First, notice that there's no `WHERE` clause this time; this is because we want to retrieve all articles, rather than an article that matches a specific ID.

We've also added a `LIMIT` clause, passing in the `$numRows` parameter (as a placeholder), so that we can optionally limit the number of records returned.

Finally, the special MySQL value `SQL_CALC_FOUND_ROWS` tells MySQL to return the actual number of records returned; this information is useful for displaying to the user, as well as for other things like pagination of results.

```
    $list = array();

    while ( $row = $st->fetch() ) {
      $article = new Article( $row );
      $list[] = $article;
    }
```

Since we're returning multiple rows, we create an array, `$list`, to hold the corresponding `Article` objects. We then use a `while` loop to retrieve the next row via `fetch()`, create a new `Article` object, store the row values in the object, and add the object to the `$list` array. When there are no more rows, `fetch()` returns `false` and the loop exits.

```
    // Now get the total number of articles that matched the criteria
    $sql = "SELECT FOUND_ROWS() AS totalRows";
    $totalRows = $conn->query( $sql )->fetch();
    $conn = null;
    return ( array ( "results" => $list, "totalRows" => $totalRows[0] ) );
```

Finally, we run another query that uses the `MySQL FOUND_ROWS()` function to get the number of returned rows calculated by our previous `SQL_CALC_FOUND_ROWS` command. This time we use the PDO `query()` method, which lets us quickly run a query if there are no placeholders to bind. We call `fetch()` on the resulting statement handle to retrieve the result row, then return both the list of `Article` objects (`$list`) and the total row count as an associative array.

6. `insert()`

The remaining methods in our `Article` class deal with adding, changing and deleting article records in the database.

`insert()` adds a new article record to the `articles` table, using the values stored in the current `Article` object:

- First, the method makes sure that the object doesn't already have its `$id` property set. If it does have an ID then the article presumably already exists in the database, so we shouldn't try to insert it again.
- Then the method runs an SQL `INSERT` query to insert the record into the `articles` table, using placeholders to pass the property values to the database. Note the use of the MySQL `FROM_UNIXTIME()` function to convert the publication date from UNIX timestamp format back into MySQL format.
- After running the query, the method retrieves the new article record's ID using the PDO `lastInsertId()` function, and stores it in the object's `$id` property for future reference. Remember that we set up the `articles` table's `id` field as an `auto_increment` field, so that MySQL generates a new unique ID for each new article record.

`PDO::PARAM_INT` when binding integer values to placeholders, and `PDO::PARAM_STR` when binding string values. This is so that PDO can escape the values appropriately.

7. `update()`

This method is similar to `insert()`, except that it updates an existing article record in the database instead of creating a new record.

First it checks that the object has an ID, since you can't update a record without knowing its ID. Then it uses the SQL `UPDATE` statement to update the record's fields. Notice that we pass the object's ID to the `UPDATE` statement so that it knows which record to update.

5. `delete()`

The `delete()` method is pretty self-explanatory. It uses the SQL `DELETE` statement to remove the article stored in the object from the `articles` table, using the object's `$id` property to identify the record in the table. For safety reasons, we add `LIMIT 1` to the query to make sure that only 1 article record can be deleted at a time.

Step 5: Write the front-end `index.php` script

We've now created our `Article` class, which does the heavy lifting for our CMS. Now that's out of the way, the rest of the code is pretty simple!

First, let's create `index.php`, the script that controls the display of the front-end pages of the site. Save this file in the `cms` folder you created earlier, at the start of Step 4:

```
<?php

require( "config.php" );
$action = isset( $_GET['action'] ) ? $_GET['action'] : "";

switch ( $action ) {
  case 'archive':
    archive();
    break;
  case 'viewArticle':
    viewArticle();
    break;
  default:
    homepage();
}


function archive() {
  $results = array();
  $data = Article::getList();
  $results['articles'] = $data['results'];
  $results['totalRows'] = $data['totalRows'];
  $results['pageTitle'] = "Article Archive | Widget News";
  require( TEMPLATE_PATH . "/archive.php" );
}

function viewArticle() {
  if ( !isset($_GET["articleId"]) || !$_GET["articleId"] ) {
    homepage();
    return;
  }

  $results = array();
  $results['article'] = Article::getById( (int)$_GET["articleId"] );
  $results['pageTitle'] = $results['article']->title . " | Widget News";
  require( TEMPLATE_PATH . "/viewArticle.php" );
}

function homepage() {
  $results = array();
  $data = Article::getList( HOMEPAGE_NUM_ARTICLES );
  $results['articles'] = $data['results'];
  $results['totalRows'] = $data['totalRows'];
  $results['pageTitle'] = "Widget News";
  require( TEMPLATE_PATH . "/homepage.php" );
}

?>
```

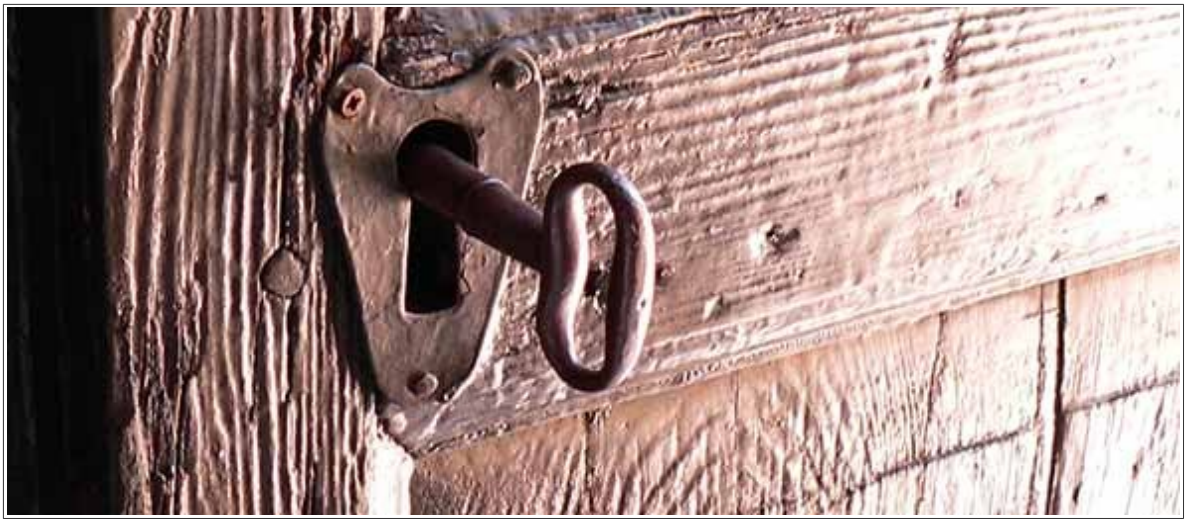
Let's break this script down:

1. **Include the config file**
   The first line of code includes the `config.php` file we created earlier, so that all the configuration settings are available to the script. We use `require()` rather than `include()`; `require()` generates an error if the file can't be found.
2. **Grab the `action` parameter**

We store the `$_GET['action']` parameter in a variable called `$action`, so that we can use the value later in the script. Before doing this, we check that the `$_GET['action']` value exists by using `isset()`. If it doesn't, we set the corresponding `$action` variable to an empty string (`""`).

3. **Decide which action to perform**
   The `switch` block looks at the `action` parameter in the URL to determine which action to perform (display the archive, or view an article). If no `action` parameter is in the URL then the script displays the site homepage.
4. **`archive()`**
   This function displays a list of all the articles in the database. It does this by calling the `getList()` method of the `Article` class that we created earlier. The function then stores the results, along with the page title, in a `$results` associative array so the template can display them in the page. Finally, it includes the template file to display the page. (We'll create the templates in a moment.)
5. **`viewArticle()`**
   This function displays a single article page. It retrieves the ID of the article to display from the `articleId` URL parameter, then calls the `Article` class's `getById()` method to retrieve the article object, which it stores in the `$results` array for the template to use. (If no `articleId` was supplied, or the article couldn't be found, then the function simply displays the homepage instead.) `(int)` to cast the value of the `articleID` query parameter to an integer. This is a good security measure, as it prevents anything other than integers from being passed to our code.
6. **`homepage()`**
   Our last function, `homepage()`, displays the site homepage containing a list of up to `HOMEPAGE_NUM_ARTICLES` articles (5 by default). It's much like the `archive()` function, except that it passes `HOMEPAGE_NUM_ARTICLES` to the `getList()` method to limit the number of articles returned.

Step 6: Write the back-end `admin.php` script



Our admin script is a bit more complex than `index.php`, since it deals with all the admin functions for the CMS. The basic structure, though, is similar to `index.php`.

Save this file, `admin.php`, in the same folder as your `index.php` script:

```
<?php

require( "config.php" );
session_start();
$action = isset( $_GET['action'] ) ? $_GET['action'] : "";
$username = isset( $_SESSION['username'] ) ? $_SESSION['username'] : "";

if ( $action != "login" && $action != "logout" && !$username ) {
  login();
  exit;
}

switch ( $action ) {
  case 'login':
    login();
    break;
  case 'logout':
    logout();
    break;
  case 'newArticle':
    newArticle();
    break;
  case 'editArticle':
    editArticle();
    break;
  case 'deleteArticle':
    deleteArticle();
    break;
  default:
    listArticles();
}


function login() {

  $results = array();
  $results['pageTitle'] = "Admin Login | Widget News";

  if ( isset( $_POST['login'] ) ) {

    // User has posted the login form: attempt to log the user in

    if ( $_POST['username'] == ADMIN_USERNAME && $_POST['password'] ==
ADMIN_PASSWORD ) {

      // Login successful: Create a session and redirect to the admin homepage
      $_SESSION['username'] = ADMIN_USERNAME;
      header( "Location: admin.php" );

    } else {

      // Login failed: display an error message to the user
      $results['errorMessage'] = "Incorrect username or password. Please try
again.";
      require( TEMPLATE_PATH . "/admin/loginForm.php" );
    }

  } else {
```

```
    // User has not posted the login form yet: display the form
    require( TEMPLATE_PATH . "/admin/loginForm.php" );
  }

}


function logout() {
  unset( $_SESSION['username'] );
  header( "Location: admin.php" );
}


function newArticle() {

  $results = array();
  $results['pageTitle'] = "New Article";
  $results['formAction'] = "newArticle";

  if ( isset( $_POST['saveChanges'] ) ) {

    // User has posted the article edit form: save the new article
    $article = new Article;
    $article->storeFormValues( $_POST );
    $article->insert();
    header( "Location: admin.php?status=changesSaved" );

  } elseif ( isset( $_POST['cancel'] ) ) {

    // User has cancelled their edits: return to the article list
    header( "Location: admin.php" );
  } else {

    // User has not posted the article edit form yet: display the form
    $results['article'] = new Article;
    require( TEMPLATE_PATH . "/admin/editArticle.php" );
  }

}


function editArticle() {

  $results = array();
  $results['pageTitle'] = "Edit Article";
  $results['formAction'] = "editArticle";

  if ( isset( $_POST['saveChanges'] ) ) {

    // User has posted the article edit form: save the article changes

    if ( !$article = Article::getById( (int)$_POST['articleId'] ) ) {
      header( "Location: admin.php?error=articleNotFound" );
      return;
    }

    $article->storeFormValues( $_POST );
    $article->update();
    header( "Location: admin.php?status=changesSaved" );

  } elseif ( isset( $_POST['cancel'] ) ) {

    // User has cancelled their edits: return to the article list
```

```
    header( "Location: admin.php" );
  } else {

    // User has not posted the article edit form yet: display the form
    $results['article'] = Article::getById( (int)$_GET['articleId'] );
    require( TEMPLATE_PATH . "/admin/editArticle.php" );
  }

}


function deleteArticle() {

  if ( !$article = Article::getById( (int)$_GET['articleId'] ) ) {
    header( "Location: admin.php?error=articleNotFound" );
    return;
  }

  $article->delete();
  header( "Location: admin.php?status=articleDeleted" );
}


function listArticles() {
  $results = array();
  $data = Article::getList();
  $results['articles'] = $data['results'];
  $results['totalRows'] = $data['totalRows'];
  $results['pageTitle'] = "All Articles";

  if ( isset( $_GET['error'] ) ) {
    if ( $_GET['error'] == "articleNotFound" ) $results['errorMessage'] =
"Error: Article not found.";
  }

  if ( isset( $_GET['status'] ) ) {
    if ( $_GET['status'] == "changesSaved" ) $results['statusMessage'] = "Your
changes have been saved.";
    if ( $_GET['status'] == "articleDeleted" ) $results['statusMessage'] =
"Article deleted.";
  }

  require( TEMPLATE_PATH . "/admin/listArticles.php" );
}

?>
```

Let’s look at some interesting sections of this script:

1. **Start a user session**

   Towards the top of the script we call `session_start()`. This PHP function starts a new session for the user, which we can use to track whether the user is logged in or not. (If a session for this user already exists, PHP automatically picks it up and uses it.)

   `session_start()` at the top of the script, before any content has been output.
2. **Grab the `action` parameter and `username` session variable**
   Next we store the `$_GET['action']` parameter in a variable called `$action`, and the `$_SESSION['username']` session variable in `$username`, so that we can use these values later in the script. Before doing this, we check that these values exist by using

`isset()`. If a value doesn't exist then we set the corresponding variable to an empty string (`""`).

3. **Check the user is logged in**
   The user shouldn't be allowed to do anything unless they're logged in as an administrator. So the next thing we do is inspect `$username` to see if the session contained a value for the `username` key, which we use to signify that the user is logged in. If `$username`'s value is empty — and the user isn't already trying to log in or out — then we display the login page and exit immediately.
4. **Decide which action to perform**

   The `switch` block works much like the one in `index.php`: it calls the appropriate function based on the value of the `action` URL parameter. The default action is to display the list of articles in the CMS.

5. **`login()`**
   This is called when the user needs to log in, or is in the process of logging in.

   If the user has submitted the login form — which we check by looking for the `login` form parameter — then the function checks the entered username and password against the config values `ADMIN_USERNAME` and `ADMIN_PASSWORD`. If they match then the `username` session key is set to the admin username, effectively logging them in, and we then redirect the browser back to the `admin.php` script, which then displays the list of articles. If the username and password don't match then the login form is redisplayed with an error message.

   If the user hasn't submitted the login form yet then the function simply displays the form.

6. **`logout()`**
   This function is called when the user elects to log out. It simply removes the `username` session key and redirects back to `admin.php`.
7. **`newArticle()`**

   This function lets the user create a new article. If the user has just posted the "new article" form then the function creates a new `Article` object, stores the form data in the object by calling `storeFormValues()`, inserts the article into the database by calling `insert()`, and redirects back to the article list, displaying a "Changes Saved" status message.

   If the user has not posted the "new article" form yet then the function creates a new empty `Article` object with no values, then uses the `editArticle.php` template to display the article edit form using this empty `Article` object.

8. **`editArticle()`**

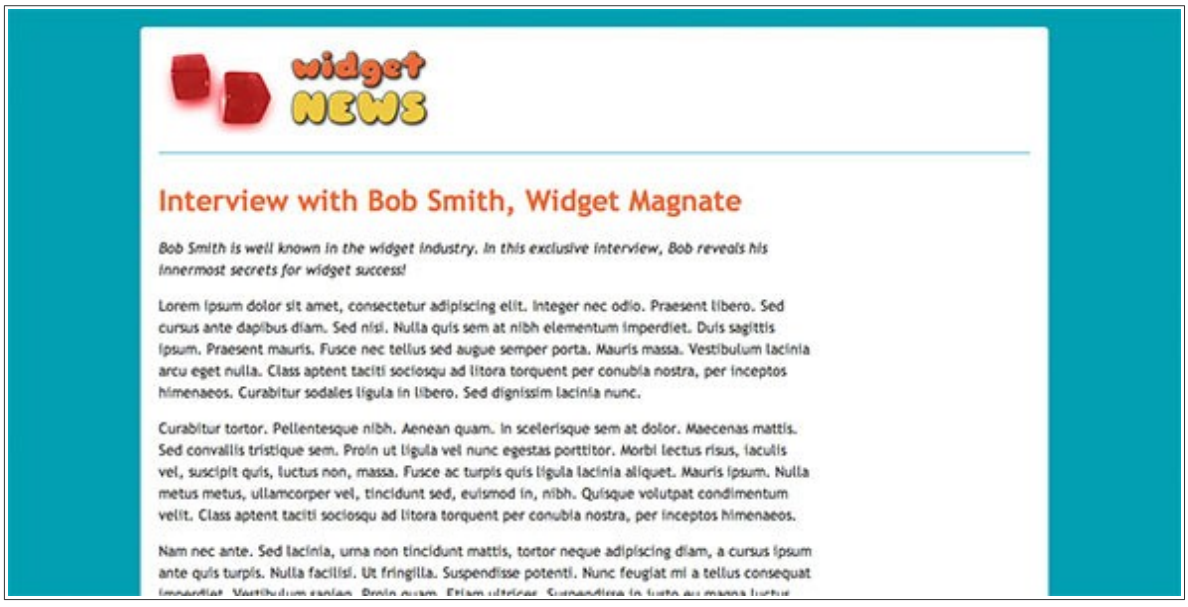
   This function is similar to `newArticle()`, except that it lets the user edit an existing article. When the user saves their changes then the function retrieves the existing article using `getById()`, stores the new values in the `Article` object, then saves the changed object by calling `update()`. (If the article isn't found in the database then the function displays an error.)

When displaying the article edit form, the function again uses the `getById()` method to load the current article field values into the form for editing.

`editArticle.php`) both for creating new articles, and for editing existing articles. This means that we only need to create a single HTML form. The `formAction` parameter is used to determine if the user is adding or editing an article.

9. **`deleteArticle()`**
   If the user has chosen to delete an article then this function first retrieves the article to be deleted (displaying an error if the article couldn't be found in the database), then calls the article's `delete()` method to remove the article from the database. It then redirects to the article list page, displaying an "article deleted" status message.
10. **`listArticles()`**
    The last function in `admin.php` displays a list of all articles in the CMS for editing. The function uses the `Article` class's `getList()` method to retrieve all the articles, then it uses the `listArticles.php` template to display the list. Along the way, it also checks the URL query parameters `error` and `status` to see if any error or status message needs to be displayed in the page. If so, then it creates the necessary message and passes it to the template for display.

Step 7: Create the front-end templates



We've now created all the PHP code for our CMS's functionality. The next step is to create the HTML templates for both the front-end and admin pages.

First, the front-end templates.

1. The include files

Create a folder called `templates` inside your `cms` folder. Now create a folder called `include` inside the `templates` folder. In this folder we're going to put the header and footer markup that is common to every page of the site, to save having to put it inside every template file.

Create a new file called `header.php` inside your `include` folder, with the following code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title><?php echo htmlspecialchars( $results['pageTitle'] )?></title>
    <link rel="stylesheet" type="text/css" href="style.css" />
  </head>
  <body>
    <div id="container">

      <a href="."><img id="logo" src="images/logo.jpg" alt="Widget News" /></a>
```

As you can see, this code simply displays the markup to start the HTML page. It uses the `$results['pageTitle']` variable passed from the main script (`index.php` or `admin.php`) to set the `title` element, and also links to a stylesheet, `style.css` (we'll create this in a moment).

`$results['pageTitle']` through the PHP function `htmlspecialchars()`. This function encodes any special HTML characters, such as <, >, and &, into their HTML entity equivalents (`<`, `>`, and `&`). Along with filtering input — which we did when we wrote the `Article` constructor in Step 4 — encoding output is a good security habit to get into. We'll encode the majority of the data in our templates this way.

Next, create a file called `footer.php` in the same folder:

```
      <div id="footer">
        Widget News &copy; 2011. All rights reserved. <a href="admin.php">Site
Admin</a>
      </div>

    </div>
  </body>
</html>
```

This markup finishes off each HTML page in the system.

2. homepage.php

Now go back up to the `templates` folder, and create a `homepage.php` template file in there, with the following code:

```
<?php include "templates/include/header.php" ?>

      <ul id="headlines">

<?php foreach ( $results['articles'] as $article ) { ?>

        <li>
          <h2>
```

```
            <span class="pubDate"><?php echo date('j F', $article-
>publicationDate)?></span><a href=".?action=viewArticle&amp;articleId=<?php echo
$article->id?>"><?php echo htmlspecialchars( $article->title )?></a>
          </h2>
          <p class="summary"><?php echo htmlspecialchars( $article->summary )?
></p>
        </li>

<?php } ?>

      </ul>

      <p><a href="./?action=archive">Article Archive</a></p>

<?php include "templates/include/footer.php" ?>
```

This template displays the article headlines on the homepage as an unordered list. It loops through the array of `Article` objects stored in `$results['articles']` and displays each article's publication date, title, and summary. The title is linked back to `'.'` (`index.php`), passing `action=viewArticle`, as well as the article's ID, in the URL. This allows the visitor to read an article by clicking its title.

The template also includes a link to the article archive (`"./?action=archive"`).

`include` statement to include the header and footer files in the page.

3. `archive.php`

Now create an `archive.php` template file in your `templates` folder:

```
<?php include "templates/include/header.php" ?>

      <h1>Article Archive</h1>

      <ul id="headlines" class="archive">

<?php foreach ( $results['articles'] as $article ) { ?>

        <li>
          <h2>
            <span class="pubDate"><?php echo date('j F Y', $article-
>publicationDate)?></span><a href=".?action=viewArticle&amp;articleId=<?php echo
$article->id?>"><?php echo htmlspecialchars( $article->title )?></a>
          </h2>
          <p class="summary"><?php echo htmlspecialchars( $article->summary )?
></p>
        </li>

<?php } ?>

      </ul>

      <p><?php echo $results['totalRows']?> article<?php echo
( $results['totalRows'] != 1 ) ? 's' : '' ?> in total.</p>

      <p><a href="./">Return to Homepage</a></p>

<?php include "templates/include/footer.php" ?>
```

This template displays the archive of all articles in the CMS. As you can see, it's almost identical to `homepage.php`. It adds an `archive` CSS class to the unordered list so we can style the list items a bit differently to the homepage, and it also adds the year to the article publication dates (since the archive might go back a few years).

The page also includes a total count of the articles in the database, retrieved via `$results['totalRows']`. Finally, instead of the archive link at the bottom of the page, it includes a "Return to Homepage" link.

4. `viewArticle.php`

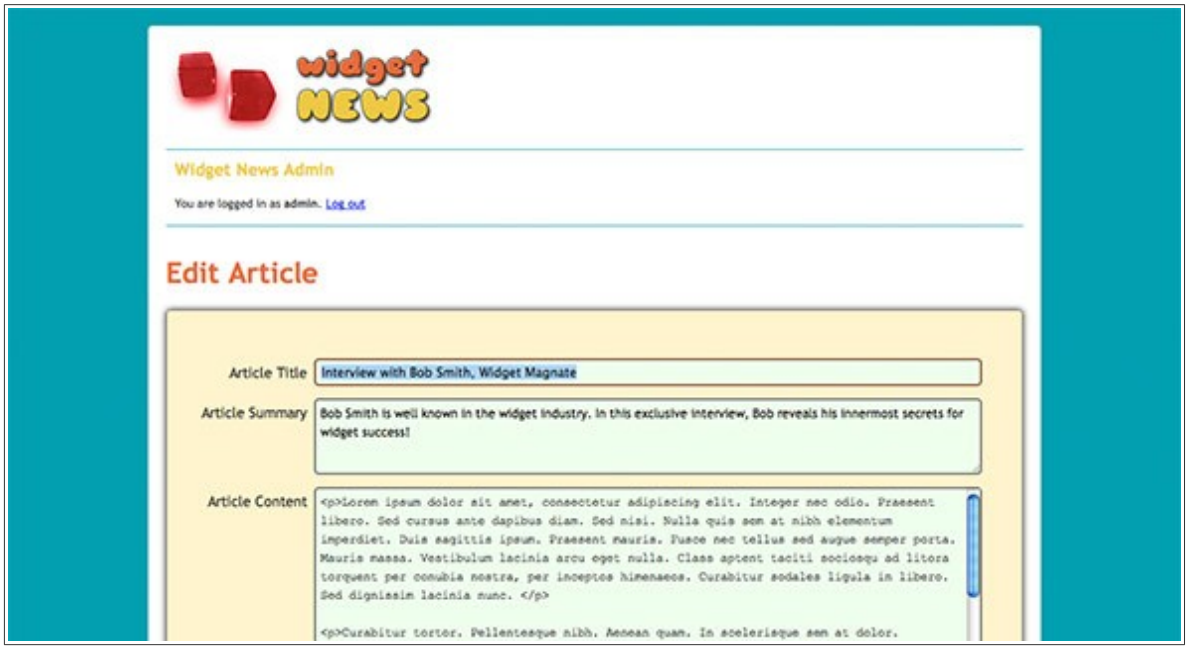
The last front-end template displays an article to the user. Create a file called `viewArticle.php` in your `templates` folder, and add the following markup:

```
<?php include "templates/include/header.php" ?>

      <h1 style="width: 75%;"><?php echo htmlspecialchars( $results['article']-
>title )?></h1>
      <div style="width: 75%; font-style: italic;"><?php echo
htmlspecialchars( $results['article']->summary )?></div>
      <div style="width: 75%;"><?php echo $results['article']->content?></div>
      <p class="pubDate">Published on <?php echo date('j F Y',
$results['article']->publicationDate)?></p>

      <p><a href="./">Return to Homepage</a></p>

<?php include "templates/include/footer.php" ?>
```

This template is very straightforward. It displays the selected article's title, summary and content, as well as its publication date and a link to return to the homepage.

haven't passed `$results['article']->content` through `htmlspecialchars()`. As explained when we created the `Article` constructor in Step 4, the administrator will probably want to use HTML markup, such as `<p>` tags, in the article content. If we encoded the content then `<p>` tags would appear on the page as `&lt;p&gt;`, rather than creating paragraphs.

Step 8: Create the back-end templates



Now that we've created the templates for the front end of the site, it's time to create the 3 admin templates.

1. `loginForm.php`

First, create another folder called `admin` inside your `templates` folder. Inside the `admin` folder, create the first of the 3 templates, `loginForm.php`:

```
<?php include "templates/include/header.php" ?>

      <form action="admin.php?action=login" method="post" style="width: 50%;">
        <input type="hidden" name="login" value="true" />

<?php if ( isset( $results['errorMessage'] ) ) { ?>
        <div class="errorMessage"><?php echo $results['errorMessage'] ?></div>
<?php } ?>

        <ul>

          <li>
            <label for="username">Username</label>
            <input type="text" name="username" id="username" placeholder="Your
admin username" required autofocus maxlength="20" />
          </li>

          <li>
            <label for="password">Password</label>
            <input type="password" name="password" id="password"
placeholder="Your admin password" required maxlength="20" />
          </li>

        </ul>

        <div class="buttons">
          <input type="submit" name="login" value="Login" />
```

```
        </div>

      </form>

<?php include "templates/include/footer.php" ?>
```

This page contains the admin login form, which posts back to `admin.php?action=login`. It includes a hidden field, `login`, that our `login()` function from Step 6 uses to check if the form has been posted. The form also contains an area for displaying any error messages (such as an incorrect username or password), as well as username and password fields and a "Login" button.

`placeholder`, `required`, `autofocus` and `date` in our admin forms. This makes the forms nicer to use, and also saves us having to check for required fields in our PHP code. Since not all browsers currently support these HTML5 form features, you would probably want to use JavaScript and/or PHP fallbacks to check for required fields in a production system.

2. `listArticles.php`

Now create the second admin template in your `admin` folder. This one's called `listArticles.php`:

```
<?php include "templates/include/header.php" ?>

      <div id="adminHeader">
        <h2>Widget News Admin</h2>
        <p>You are logged in as <b><?php echo
htmlspecialchars( $_SESSION['username']) ?></b>. <a href="admin.php?
action=logout"?>Log out</a></p>
      </div>

      <h1>All Articles</h1>

<?php if ( isset( $results['errorMessage'] ) ) { ?>
        <div class="errorMessage"><?php echo $results['errorMessage'] ?></div>
<?php } ?>


<?php if ( isset( $results['statusMessage'] ) ) { ?>
        <div class="statusMessage"><?php echo $results['statusMessage'] ?></div>
<?php } ?>

      <table>
        <tr>
          <th>Publication Date</th>
          <th>Article</th>
        </tr>

<?php foreach ( $results['articles'] as $article ) { ?>

        <tr onclick="location='admin.php?action=editArticle&amp;articleId=<?php
echo $article->id?>'">
          <td><?php echo date('j M Y', $article->publicationDate)?></td>
          <td>
            <?php echo $article->title?>
          </td>
        </tr>

<?php } ?>
```

```
      </table>

      <p><?php echo $results['totalRows']?> article<?php echo
( $results['totalRows'] != 1 ) ? 's' : '' ?> in total.</p>

      <p><a href="admin.php?action=newArticle">Add a New Article</a></p>

<?php include "templates/include/footer.php" ?>
```

This template displays the list of articles for the administrator to edit. After displaying any error or status messages, it loops through the array of `Article` objects stored in `$results['articles']`, displaying each article's publication date and title in a table row. It also adds a JavaScript `onclick` event to each article's table row, so that the administrator can click an article to edit it.

The template also includes the total article count, as well as a link to let the administrator add a new article.

3. `editArticle.php`

Now save the final template, `editArticle.php`, in your `admin` folder:

```
<?php include "templates/include/header.php" ?>

      <div id="adminHeader">
        <h2>Widget News Admin</h2>
        <p>You are logged in as <b><?php echo
htmlspecialchars( $_SESSION['username']) ?></b>. <a href="admin.php?
action=logout"?>Log out</a></p>
      </div>

      <h1><?php echo $results['pageTitle']?></h1>

      <form action="admin.php?action=<?php echo $results['formAction']?>"
method="post">
        <input type="hidden" name="articleId" value="<?php echo
$results['article']->id ?>"/>

<?php if ( isset( $results['errorMessage'] ) ) { ?>
        <div class="errorMessage"><?php echo $results['errorMessage'] ?></div>
<?php } ?>

        <ul>

          <li>
            <label for="title">Article Title</label>
            <input type="text" name="title" id="title" placeholder="Name of the
article" required autofocus maxlength="255" value="<?php echo
htmlspecialchars( $results['article']->title )?>" />
          </li>

          <li>
            <label for="summary">Article Summary</label>
            <textarea name="summary" id="summary" placeholder="Brief description
of the article" required maxlength="1000" style="height: 5em;"><?php echo
htmlspecialchars( $results['article']->summary )?></textarea>
          </li>

          <li>
            <label for="content">Article Content</label>
```

```
            <textarea name="content" id="content" placeholder="The HTML content
of the article" required maxlength="100000" style="height: 30em;"><?php echo
htmlspecialchars( $results['article']->content )?></textarea>
          </li>

          <li>
            <label for="publicationDate">Publication Date</label>
            <input type="date" name="publicationDate" id="publicationDate"
placeholder="YYYY-MM-DD" required maxlength="10" value="<?php echo
$results['article']->publicationDate ? date( "Y-m-d", $results['article']-
>publicationDate ) : "" ?>" />
          </li>


        </ul>

        <div class="buttons">
          <input type="submit" name="saveChanges" value="Save Changes" />
          <input type="submit" formnovalidate name="cancel" value="Cancel" />
        </div>

      </form>

<?php if ( $results['article']->id ) { ?>
      <p><a href="admin.php?action=deleteArticle&amp;articleId=<?php echo
$results['article']->id ?>" onclick="return confirm('Delete This
Article?')">Delete This Article</a></p>
<?php } ?>

<?php include "templates/include/footer.php" ?>
```

This edit form is used both for creating new articles, and for editing existing articles. It posts to either `admin.php?action=newArticle` or `admin.php?action=editArticle`, depending on the value passed in the `$results['formAction']` variable. It also contains a hidden field, `articleId`, to track the ID of the article being edited (if any).

The form also includes an area for error messages, as well as fields for the article title, summary, content, and publication date. Finally, there are 2 buttons for saving and cancelling changes, and a link to allow the admin to delete the currently-edited article.

`htmlspecialchars()` before outputting it in the markup. Not only is this a good security habit, but it also ensures that our form field values are properly escaped. For example, if the `title` field value contained a double quote (") that wasn't escaped then the title would be truncated, since double quotes are used to delimit the field's value in the markup. `formnovalidate` attribute on the "Cancel" button. This handy attribute tells the browser not to validate the form if the user presses "Cancel".

Step 9: Create the stylesheet and logo image

Our CMS application is basically done now, but in order to make it look a bit nicer for both our visitors and the site administrator, we'll create a CSS file to control the look of the site. Save this file as `style.css` in your `cms` folder:

```
/* Style the body and outer container */

body {
  margin: 0;
```

```
  color: #333;
  background-color: #00a0b0;
  font-family: "Trebuchet MS", Arial, Helvetica, sans-serif;
  line-height: 1.5em;
}

#container {
  width: 960px;
  background: #fff;
  margin: 20px auto;
  padding: 20px;
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
}


/* The logo and footer */

#logo {
  display: block;
  width: 300px;
  padding: 0 660px 20px 0;
  border: none;
  border-bottom: 1px solid #00a0b0;
  margin-bottom: 40px;
}

#footer {
  border-top: 1px solid #00a0b0;
  margin-top: 40px;
  padding: 20px 0 0 0;
  font-size: .8em;
}


/* Headings */

h1 {
  color: #eb6841;
  margin-bottom: 30px;
  line-height: 1.2em;
}

h2, h2 a {
  color: #edc951;
}

h2 a {
  text-decoration: none;
}


/* Article headlines */

#headlines {
  list-style: none;
  padding-left: 0;
  width: 75%;
}

#headlines li {
  margin-bottom: 2em;
}
```

```
.pubDate {
  font-size: .8em;
  color: #eb6841;
  text-transform: uppercase;
}

#headlines .pubDate {
  display: inline-block;
  width: 100px;
  font-size: .5em;
  vertical-align: middle;
}

#headlines.archive .pubDate {
  width: 130px;
}

.summary {
  padding-left: 100px;
}

#headlines.archive .summary {
  padding-left: 130px;
}


/* "You are logged in..." header on admin pages */

#adminHeader {
  width: 940px;
  padding: 0 10px;
  border-bottom: 1px solid #00a0b0;
  margin: -30px 0 40px 0;
  font-size: 0.8em;
}


/* Style the form with a coloured background, along with curved corners and a
drop shadow */

form {
  margin: 20px auto;
  padding: 40px 20px;
  overflow: auto;
  background: #fff4cf;
  border: 1px solid #666;
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
  -moz-box-shadow: 0 0 .5em rgba(0, 0, 0, .8);
  -webkit-box-shadow: 0 0 .5em rgba(0, 0, 0, .8);
  box-shadow: 0 0 .5em rgba(0, 0, 0, .8);
}


/* Give form elements consistent margin, padding and line height */

form ul {
  list-style: none;
  margin: 0;
  padding: 0;
}
```

```
form ul li {
  margin: .9em 0 0 0;
  padding: 0;
}

form * {
  line-height: 1em;
}


/* The field labels */

label {
  display: block;
  float: left;
  clear: left;
  text-align: right;
  width: 15%;
  padding: .4em 0 0 0;
  margin: .15em .5em 0 0;
}


/* The fields */

input, select, textarea {
  display: block;
  margin: 0;
  padding: .4em;
  width: 80%;
}

input, textarea, .date {
  border: 2px solid #666;
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
  background: #fff;
}

input {
  font-size: .9em;
}

select {
  padding: 0;
  margin-bottom: 2.5em;
  position: relative;
  top: .7em;
}

textarea {
  font-family: "Trebuchet MS", Arial, Helvetica, sans-serif;
  font-size: .9em;
  height: 5em;
  line-height: 1.5em;
}

textarea#content {
  font-family: "Courier New", courier, fixed;
}


/* Place a border around focused fields */
```

```
form *:focus {
  border: 2px solid #7c412b;
  outline: none;
}


/* Display correctly filled-in fields with a green background */

input:valid, textarea:valid {
  background: #efe;
}


/* Submit buttons */

.buttons {
  text-align: center;
  margin: 40px 0 0 0;
}

input[type="submit"] {
  display: inline;
  margin: 0 20px;
  width: 12em;
  padding: 10px;
  border: 2px solid #7c412b;
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
  -moz-box-shadow: 0 0 .5em rgba(0, 0, 0, .8);
  -webkit-box-shadow: 0 0 .5em rgba(0, 0, 0, .8);
  box-shadow: 0 0 .5em rgba(0, 0, 0, .8);
  color: #fff;
  background: #ef7d50;
  font-weight: bold;
  -webkit-appearance: none;
}

input[type="submit"]:hover, input[type="submit"]:active {
  cursor: pointer;
  background: #fff;
  color: #ef7d50;
}

input[type="submit"]:active {
  background: #eee;
  -moz-box-shadow: 0 0 .5em rgba(0, 0, 0, .8) inset;
  -webkit-box-shadow: 0 0 .5em rgba(0, 0, 0, .8) inset;
  box-shadow: 0 0 .5em rgba(0, 0, 0, .8) inset;
}


/* Tables */

table {
  width: 100%;
  border-collapse: collapse;
}

tr, th, td {
  padding: 10px;
  margin: 0;
  text-align: left;
```

```
}

table, th {
  border: 1px solid #00a0b0;
}

th {
  border-left: none;
  border-right: none;
  background: #ef7d50;
  color: #fff;
  cursor: default;
}

tr:nth-child(odd) {
  background: #fff4cf;
}

tr:nth-child(even) {
  background: #fff;
}

tr:hover {
  background: #ddd;
  cursor: pointer;
}


/* Status and error boxes */

.statusMessage, .errorMessage {
  font-size: .8em;
  padding: .5em;
  margin: 2em 0;
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  border-radius: 5px;
  -moz-box-shadow: 0 0 .5em rgba(0, 0, 0, .8);
  -webkit-box-shadow: 0 0 .5em rgba(0, 0, 0, .8);
  -box-shadow: 0 0 .5em rgba(0, 0, 0, .8);
}

.statusMessage {
  background-color: #2b2;
  border: 1px solid #080;
  color: #fff;
}

.errorMessage {
  background-color: #f22;
  border: 1px solid #800;
  color: #fff;
}
```

I won't go into the details of the CSS, since this tutorial is about PHP and MySQL! Suffice to say, it styles things like the page layout, colours, fonts, forms, tables and so on.

Last, but not least, our site needs a logo. Here's one I prepared earlier — save it in an `images` folder inside your `cms` folder, calling it `logo.jpg` (or roll your own logo):

All done!

We've finished our CMS! To try it out, open a browser and point it to the base URL of your CMS (for example, `http://localhost/cms/`). Click the em>Site Admin link in the footer, log in, and add some articles. Then try browsing them on the front end (click the logo to return to the homepage).

Don't forget you can try out the demo on my server too!

In this tutorial you've built a basic content management system from the ground up, using PHP and MySQL. You've learnt about MySQL, tables, field types, PDO, object-oriented programming, templating, security, sessions, and lots more.

While this CMS is pretty basic, it has hopefully given you a starting point for building your own CMS-driven websites. Some features you might want to add include:

- Pagination on the article archive (front end) and article list (back end) so that the system can easily handle hundreds of articles
- A WYSIWYG editor for easier content editing
- An image upload facility (I've written a follow-up tutorial on adding an image upload feature to the CMS)
- A preview facility, so the admin can see how an article will look before publishing it
- Article categories and tags (I've written a follow-up tutorial on adding categories)
- Integration with Apache's mod_rewrite to create more human-friendly permalink URLs for the articles (find out how to do this)
- A user comments system

I hope you've enjoyed this tutorial and found it useful. Happy coding!

Filed Under: PHP Tagged With: cms, mysql, php, Tutorial, web development

https://www.elated.com/cms-in-an-afternoon-php-mysql/