We'll populate the figure with the relevant content. First, however, we need to see whether there's an image to be displayed:

```
<?php
#   blog.code.php
    ...

    $article = pilcrow2p($article,true);

    //  Optional Image
        if($imageid) {

        }
```

If there is an image, we can define the `$figure` variable as a template string:

```
if($imageid) {
    $figure = '<figure>
        <a href="/images/scaled/%s">
            <img src="/images/thumbnails/%s" width="%s"
            height="%s"
                alt="%s">
        </a>
        <figcaption>%s</figcaption>
    </figure>';
}
```

As usual, the string can be written on multiple lines and needs to be to fit in this book. You can keep it all on one line if you prefer.

We'll get the image from the `thumbnails` directory, and so we'll need to get the image dimensions for that size. That's in the `$CONFIG` array. The image is wrapped in an anchor which will be used to display a larger version from the `scaled` directory.

The image file and text will come from the `images` table. To get the data, we'll use the `$imageid` variable to read from the database:

```
//  Optional Image
    if($imageid) {
        $figure = ... ;

        $sql = "SELECT title, src FROM images WHERE
        id=$imageid";
        [$title, $src] = $pdo -> query($sql) -> fetch();

    }
```

Again, the value in `$imageid` comes from a safe source—in this case, it's the `blog` table—so we can simply interpolate into the SQL statement without the need to prepare it. From there, we can run the query and fetch the result. We can destructure the result directly into the variables `$title` and `$src`.

The `img` inside the `figure` string incudes placeholders for the width and height of the image. We'll use the `splitSize()` function in the default library to split the data from the `$CONFIG` array:

```
//  Optional Image
    if($imageid) {
        $figure = ... ;

        $sql = "SELECT title, src FROM images WHERE
        id=$imageid";
        [$title, $src] = $pdo -> query($sql) -> fetch();
        [$width, $height]
            = splitSize($CONFIG['images']['thumbnail-size']);

    }
```

We can now use the `sprintf()` function to put the values into the `$figure` string:

```
//  Optional Image
    if($imageid) {
        $figure = ... ;
        $sql = "SELECT title, src FROM images WHERE
        id=$imageid";
        [$title, $src] = $pdo -> query($sql) -> fetch();
        [$width, $height]
            = splitSize($CONFIG['images']['thumbnail-size']);

        $figure = sprintf(
            $figure,
            $src, $src, $width, $height, $title, $title
        );
    }
```

The `$src` variable is used twice: once for the enclosing anchor and once for the image itself. The `$title` is also used twice: once for the `alt` attribute and once for the figure caption.

You can now test various article IDs, but you'll have to enter them manually:

```
http://australia.example.com/blogarticle.php?article=4
```

Some of the articles will include an image, while some won't. If you include an article ID which you know will be out of range, then you'll be redirected to the `bloglist.php` page which is supposed to be a list of blog articles for visitors; at the moment, it's empty.

We'll work on the `bloglist.php` page a little later. First, we'll work on the blog management section.

Managing the Blog Articles

You've now got a collection of blog articles, and you can now view them. Here, we'll develop the management section.

The blog management pages will work similarly to the image management pages we worked on in Chapter 9:

- There will be a list page called `admin-bloglist.php`, which will list the blog articles for editing and have buttons to edit, delete, or add articles.
- The list page will lead to an edit page called `editblog.php`, where the individual article will be edited, deleted, or added. We've already worked on the code to add an article on this page.
- Between the `admin-bloglist.php` and `editblog.php` pages, there will be various submit events, and the `editblog.php` will also have submit events to confirm changes.

The article list page will look something like Figure 12-5.

Figure 12-5. *Administration List Page*

First, we'll create another file in the `includes` directory called `admin-bloglist.code.php`. We can begin the code with the preliminaries, such as we put in the `imagelist.code.php` file, without, of course, closing the PHP tag:

```
<?php
#    admin-bloglist.code.php
```

```
if(!session_id()) {
    session_start();
    session_regenerate_id(true);
}

$root = str_replace($_SERVER['SCRIPT_NAME'], '',
    $_SERVER['SCRIPT_FILENAME']);
$host = $_SERVER['HTTP_HOST'];
$protocol = isset($_SERVER['HTTPS']) &&
$_SERVER['HTTPS'] == 'on'
    ? 'https'
    : 'http';

if(!isset($_SESSION['user']) || !$_SESSION['admin']) {
    header("Location: $protocol://$host/admin.php");
    exit;
}

$pdo = require_once "$root/includes/db.php";
require_once "$root/includes/library.php";
require_once "$root/includes/default-library.php";

$CONFIG = parse_ini_file("$root/config.ini.php",true);
```

In the admin-bloglist.php page is a PHP block which initializes two variables. We can move that code to the admin-bloglist.code.php after the rest of the code:

```
<?php
#   admin-bloglist.code.php
    ...
    $CONFIG = parse_ini_file("$root/config.ini.php",true);

    $tbody = $paging = $displaying = '';
```

Meanwhile, at the beginning of the admin-bloglist.php page, replace the removed code with an include:

```
<?php require_once 'includes/admin-bloglist.code.php'; ?>
<?php
    $pagetitle = 'Administration Blog List';
    $pageheading = 'Blog List';

    require_once 'includes/head.inc.php';
?>
...
```

The blog list page will cater for a large number of articles, the same way the image list and the gallery pages, by paging the results. As with the image list, we'll put the number of articles per page in the config.ini.php file:

```
...
[imagelist]
    page-size = 6
[admin-bloglist]
    page-size = 4
```

The blog list page can fit many more than four items, of course, and you can experiment with a larger number when the code is finished. It's set to such a low number simply to test the paging.

To process the page number, we can copy the code from the imagelist.code.php file:

```
<?php
#   admin-bloglist.code.php
    ...

    $tbody = $paging = displaying = '';
```

```
$page = intval($_GET['page'] ?? $_COOKIE
['admin-bloglist-page']
    ?? 1) ?: 1;

$limit = $CONFIG['admin-bloglist']['page-size'];
$offset = ($page - 1) * $limit;

$blogCount = $pdo -> query('SELECT count(*) FROM blog')
    -> fetchColumn();
$pages = ceil($blogCount / $limit);

$page = max(1, $page);          //  $page is at least 1
$page = min($page, $pages);     //  $page is up to $pages
setcookie('admin-bloglist-page', $page, strtotime
('+ 1 hour'));

$paging = paging($page, $pages);
$displaying = "Page $page of $pages";
```

Again, we've made a number of changes (six, if you count them):

- The cookie name is `admin-bloglist-page`. We've changed that for the `$_COOKIE` array and the `setcookie()` function.
- The `$CONFIG` array references `admin-bloglist`.
- The `SELECT` statement selects from the `blog` table.
- The variable name for the number of rows is changed to `$blogCount`, both for the PDO query and the `ceil()` calculation on the next line.
- We've added the `$displaying` variable.

We can now build the table of blog articles.

Building the Article Table

The table will have the following columns:

ID	Title	Created	Updated	[Edit] [Delete]

We can begin with the template strings, copied from the `imagelist.code.php` page, with a few changes:

```
<?php
#   admin-bloglist.code.php
    ...

    setcookie('admin-bloglist-page', $page, strtotime('+ 1
    hour'));

    $tr = '<tr><th>%s</th><td>%s</td><td>%s</td><td>%s</td>
        <td>%s</td><td>%s</td></tr>';

    $editButton = '<button name="prepare-update"
        value="%s">Edit</button>';
    $deleteButton = '<button name="prepare-delete"
        value="%s">Delete</button>';

    $paging = paging($page, $pages);
    $displaying = "Page $page of $pages";
```

We've made some changes, of course:

- We don't need the image strings and data which we used in the `imagelist.code.php` page.
- The `tr` string has *six* cells.

  We've also changed the first one to a `th` cell for good measure, but it's not necessary.

Also note that the string wraps to another line because it doesn't quite fit in this book. If possible, you should try to keep the string on one line.

- We've changed the text of the `delete` button to **Delete** rather than **Remove** because it reads better; we've also changed the name of the variable.

We can now write the SQL query and iterate through the results. As before, we'll put the table rows into an array called `$tr`:

```
<?php
#   admin-bloglist.code.php
    ...

    $editButton = '<button name="prepare-update"
        value="%s">Edit</button>';
    $deleteButton = '<button name="prepare-delete"
        value="%s">Delete</button>';

    $sql="SELECT id, title, created, updated FROM blog
        ORDER BY id LIMIT $limit OFFSET $offset";

    $tbody = [];
    foreach($pdo->query($sql) as [$id, $title, $created,
    $updated]) {

    }
    $tbody = implode($tbody);

    $paging = paging($page, $pages);
    $displaying = "Page $page of $pages";
```

The next step is to populate the table rows. This will require the following steps:

- Format the date using a combination of the `date()` and `strtotime()` functions.
- The date format is `d M Y g:i a` which will give a date and time like `20 Jul 69 8:17 pm`.
- Populate the `$edit` and `$delete` values using `sprintf()`.
- Add a new row to the `$tbody` array, using the `sprintf()` function on the `$tr` template string.

This gives us

```
<?php
#   admin-bloglist.code.php
    ...

    $tbody = [];
    foreach($pdo->query($sql) as [$id, $title, $created,
    $updated]) {
        $created = date('d M Y g:i a', strtotime($created));
        $updated = date('d M Y g:i a', strtotime($updated));
        $edit = sprintf($editButton, $id);
        $delete = sprintf($deleteButton, $id);
        $tbody[] = sprintf($tr, $id, $title, $created,
        $updated,
            $edit, $delete);
    }
    $tbody = implode($tbody);

    ...
```

We now have a Blog List page which will list the blog articles.

We'll now process the **Edit** and **Delete** buttons, which will prepare the `editblog.php` page. This is done in the `manage-blog.code.php` file.

Processing the Prepare Events

In the `manage-blog.code.php` page, we have already written the code to prepare for a new blog article. We can now remove the `empty($_POST) ||` part, since we've tested the code and we'll now presume that we're coming from the `admin-bloglist.php` page, where there's a button for this:

```
<?php
#   manage-blog.code.php
    ...

    //  Prepare Page
        if(isset($_POST['prepare-insert'])) {
            $pagetitle = 'Upload Article';
            $pageheading = 'Upload Article';

            $chooseImage = getImageSelect();
        }
```

We can begin by adding blocks to prepare for the update and delete events. This is similar to the code in the `manage-images.code.php` file, where we first set the title and heading for the page:

```
<?php
#   manage-blog.code.php
    ...

    //  Prepare Page
        if(isset($_POST['prepare-insert'])) {
            ...
        }
```

```
        if(isset($_POST['prepare-update'])) {
            $pagetitle = 'Edit Article';
            $pageheading = 'Edit Article';

        }

        if(isset($_POST['prepare-delete'])) {
            $pagetitle = 'Delete Article';
            $pageheading = 'Delete Article';

        }
```

Again, most of the following code is modelled after `manage-images.code.php` file. First, we'll extract the ID from the submit button:

```
<?php
#   manage-blog.code.php
    ...

    //  Prepare Page
        if(isset($_POST['prepare-insert'])) {
            ...
        }

        if(isset($_POST['prepare-update'])) {
            $pagetitle = 'Edit Article';
            $pageheading = 'Edit Article';

            $id = intval($_POST['prepare-update'] ?? 0);
        }


        if(isset($_POST['prepare-delete'])) {
            $pagetitle = 'Delete Article';
            $pageheading = 'Delete Article';

            $id = intval($_POST['prepare-delete'] ?? 0);

        }
```

A missing `id` will default to `0`, which is no article.

At this point, we'll need to fetch the article data and possible image for both events. It would be easier to do that in a function so we don't have to repeat the code.

In the function section of the code, add the following new function:

```
// Functions
    function getImageSelect() {
        ...
    }
    function getImageButtons() {
        ...
    }

    function getArticle(int $id) {

    }
```

The only input parameter will be the article `id`. The return result, however, will be a whole array of data.

First, we'll get the article data from the database. Among other things, that means we'll need the `$pdo` object. We'll also include the `$CONFIG` array for the image:

```
function getArticle(int $id) {
    global $pdo, $CONFIG;

    $sql = "SELECT title, precis, created, updated,
    article, imageid
        FROM blog WHERE id=$id";
    $row = $pdo -> query($sql) -> fetch();
}
```

It's possible that the article ID, even if not zero, is invalid—possibly it refers to a deleted article. If it's invalid, then, of course, the `$row` variable will be `null`, and we can return immediately with a `null`. Otherwise, we can continue and extract the data into variables. We can also process the extracted `article` data through the `pilcrow2nl()` function and format the two dates:

```
function getArticle(int $id) {
    global $pdo, $CONFIG;

    $sql = "SELECT title, precis, created, updated,
    article, imageid
        FROM blog WHERE id=$id";
    $row = $pdo -> query($sql) -> fetch();

    if(!$row) return null;      //  exit if no data

    [$title, $precis, $created, $updated, $article,
    $imageid] = $row;
    $article = pilcrow2nl($article);
    $created = date('d M Y g:i a', strtotime($created));
    $updated = date('d M Y g:i a', strtotime($updated));
}
```

The dates are processed the same way as they were for the blog article.

Displaying the Image

There may also be an image for the article. We can test the `$imageid` variable and, if so, fetch the data from the `images` table. If not, we'll generate a blank image in its place.

We already have the $previewImage template string, which we'll need to import. We'll also need to get the dimensions for the thumbnail size:

```
function getArticle(int $id) {
    global $pdo, $CONFIG, $previewImage;

    ...

    $created = date('d M Y g:i a', strtotime($created));
    $updated = date('d M Y g:i a', strtotime($updated));

    [$width, $height]
        = splitSize($CONFIG['images']['thumbnail-size']);
}
```

We can now test for an image and, if so, fetch the data from the images table. Otherwise, we'll set the image to a blank image:

```
function getArticle(int $id) {
    global $pdo, $CONFIG, $previewImage;
    ...

    [$width, $height]
        = splitSize($CONFIG['images']['thumbnail-size']);

    if($imageid) {

    }
    else $previewImage = sprintf(
        $previewImage,
        '/images/blank.png', $width, $height, 'No Image'
    );
}
```

Notice that we're rewriting the $previewImage variable—we only need the original value long enough to use in the sprintf() function, and the new value actually holds the preview image.

If there is an image, we'll fetch it and test whether that was successful:

```
function getArticle(int $id) {
    ...

    if($imageid) {
        $sql = "SELECT title, src FROM images WHERE
        id=$imageid";
        $row = $pdo -> query($sql) -> fetch();
        if($row) {
            [$imagetitle, $src] = $row;
            $previewImage = sprintf(
                $previewImage,
                "images/thumbnails/$src", $width, $height,
                $imagetitle
            );
        }
    }
    else $previewImage = sprintf( ... );
}
```

Note that we've interpolated the `src` variable into the string to include the images directory.

Note also that we've used the variable `$imagetitle` for the image title, rather than just `$title`. That's because the `$title` variable is being used for the article title. We don't need the `$imagetitle` variable for very long, as it's only being used to generate the `$previewImage` value.

Finally, we can return the data and the image in an array:

```
function getArticle(int $id) {
    ...
```

```
    if($imageid) {
       ...
    }
    else $image = sprintf( ... );

    return [$title, $precis, $created, $updated, $article,
        $imageid, $previewImage];
}
```

Apart from the text data for the article, the return array also includes the ID of the image, of any, and the preview image.

We should now fetch the data into variables in the prepare update and prepare delete code:

```
if(isset($_POST['prepare-update'])) {
    ...

    $id = intval($_POST['prepare-update'] ?? 0);
    [$title, $precis, $created, $updated, $article, $imageid,
        $previewImage] = getArticle($id);
}

if(isset($_POST['prepare-delete'])) {
    ...

    $id = intval($_POST['prepare-delete'] ?? 0);
    [$title, $precis, $created, $updated, $article, $imageid,
        $previewImage] = getArticle($id);
}
```

Finishing the Preparation

There are two more tasks to finish the preparation.

First, for the prepare update, we'll include the option to select a preexisting image to replace the current image. However, we should highlight the currently selected image if there is one. The form already includes the option to select a new image.

For both the `getImageSelect()` and the `getImageButtons()` functions, add an optional parameter variable for the current image:

```
function getImageSelect($imageid=0) {
    ...
}

function getImageButtons($imageid=0) {
    ...
}
```

The default value is `0` which means that there isn't one.

In a `select` menu, to preselect an `option` element you set its `selected` attribute. We already have a placeholder in the template string, and currently, it's set to an empty string.

The first option will always be selected if nothing else is. However, if another one is selected, then it will replace the first option.

Inside the `foreach` block, we can set the value to either `selected` or an empty string, depending on whether the current `$id` matches the `$imageid` variable. This is where the ternary operator allows to choose one of two values:

```
function getImageSelect($imageid=0) {
    ...

    foreach($pdo -> query('SELECT id, title, src FROM images')
        AS [$id, $title, $src]) {
        $select[] =sprintf(
            $option, $id, "/images/thumbnails/$src",
            $id == $imageid ? ' selected' : '', $id, $title
        );
```

```
    }

    ...
}
```

For the image buttons, the idea is the same, except that we set the checked attribute of the selected radio button:

```
function getImageButtons($imageid=0) {
    ...

    foreach($pdo -> query('SELECT id, title, src FROM images')
        as [$id, $title, $src]) {
        $buttons[] = sprintf(
            $button, $id, $id == $imageid ? ' checked' : '',
            "/images/icons/$src", $title, $title,
            $width, $height, "/images/thumbnails/$src"
        );
    }
    ...
}
```

In the prepare update code, we can add the call to the `getImageSelect()` or `getImageButtons()` function, complete with the `$imageid` of the current article:

```
if(isset($_POST['prepare-update'])) {
    ...

    [$title, $precis, $created, $updated, $article, $imageid,
        $previewImage] = getArticle($id);

    $chooseImage = $CONFIG['blogedit']['use-image']=='buttons'
        ? getImageButtons($imageid)
        : getImageSelect($imageid);
}
```

For the prepare delete, we won't allow the user to choose an image. What we will do, however, is set the `$disabled` variable to disable editing on the form:

```
if(isset($_POST['prepare-delete'])) {
    ...
    $disabled = ' disabled';
}
```

We can now test the blog list. Of course, any changes you make at this point will be ignored, until we finish writing the rest of the editing code.

Updating an Article

We've already worked on the code to update an image. Updating a blog article will be similar.

To begin with, we'll add a block for the update:

```
    ...
    if(isset($_POST['insert'])) {
        ...
    }
    if(isset($_POST['update'])) {
    }

    if(isset($_POST['import'])) {
        ...
    }
    ...
```

To get the data ready for uploading, we'll use similar code for the insert to fetch the data and test for errors:

```
if(isset($_POST['update'])) {
    [$id, $title, $precis, $article, $image, $errors] =
    getBlogData();
    $precis = nl2pilcrow($precis);
    $article = nl2pilcrow($article);

    if(!$errors) {  //  proceed

        //  Move On
            header("Location: $protocol://$host/admin-
            bloglist.php");
            exit;
    }
    else {          //  handle error
        $errors = sprintf('<p class="errors">%s</p>',
            implode('<br>',$errors));
    }
}
```

Note that we've also put the `$precis` and `$article` variables through the `nl2pilcrow()` function. That's because we *won't* use the `addBlogData()` function. Instead, we can just write the necessary code directly.

To update an article, you use the `UPDATE` statement:

```
UPDATE blog
SET title=?, precis=?, article=?, updated=?
WHERE id=?
```

The text content is, of course, included. So is the `updated` column, which is why it's there. There's also the image, but we'll deal with that later.

We'll write that into the PHP code:

```
if(isset($_POST['update'])) {
    ...

    if(!$errors) {  //  proceed
        $sql = 'UPDATE blog SET title=?, precis=?, article=?,
            updated=? WHERE id=?';
        $pdoStatement = $pdo -> prepare($sql);
        $data = [$title, $precis, $article, date
        ('Y-m-d H:i:s'), $id];
        $pdoStatement->execute($data);

        //  Move On
            ...
    }
    ...
}
```

The `updated` column will get the formatted current date/time.

If there's an image to be included in the updated data, it should also be added. This is why it was important to select the current image in the choose image section.

To include the image, we can just call the `addBlogImage()` function, which does all of the hard work:

```
if(isset($_POST['update'])) {
    ...
```

```
    if(!$errors) {  //  proceed
        $sql = 'UPDATE blog SET title=?, precis=?, article=?,
            updated=? WHERE id=?';
        $pdoStatement = $pdo -> prepare($sql);
        $data = [$title, $precis, $article,
        date('Y-m-d H:i:s'), $id];
        $pdoStatement->execute($data);

        addBlogImage($id, $image, $title, $precis);

        //  Move On
            ...
    }
    ...
}
```

You can now test this by starting on the blog list page, selecting an article to edit, and making a few minor or major changes. If you revisit the article for editing, you should see all of your changes.

Deleting an Article

This one is pretty straightforward, except when it comes to images. While we're happy to add a new image to a blog article, it's not easy to guess whether we want to keep the image later, possibly as part of the image gallery. The simple solution is to leave that to the Image Management section.

We'll begin by creating the delete block:

```
...
if(isset($_POST['update'])) {
    ...
```

```
}

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

}

if(isset($_POST['import'])) {
    ...
}

...
```

To delete an article, we use the DELETE statement:

```
DELETE FROM blog WHERE id=?
```

We have the getBlogData() function to retrieve the uploaded data, but all we really need is the ID. We can get that directly from the $_POST array:

```
if(isset($_POST['delete'])) {
    $id = intval($_POST['id'] ?? 0);
}
```

Since the $id variable is guaranteed to be an integer, we can interpolate it into an SQL string and run it directly using the exec() method:

```
if(isset($_POST['delete'])) {
    $id = intval($_POST['id'] ?? 0);

    $pdo -> exec("DELETE FROM images WHERE id=$id");
}
```

After that, we can move on to the blog list:

```
if(isset($_POST['delete'])) {
    $id = intval($_POST['id'] ?? 0);

    $pdo -> exec("DELETE FROM images WHERE id=$id");
```

```
    //  Move On
        header("Location: $protocol://$host/admin-
        bloglist.php");
        exit;
}
```

We can now try deleting an unwanted blog article.

The Visitor Blog List

Everything is almost in place. The one thing missing is the blog list page for visitors. However, that's easily developed with the help of the code we developed for the admin blog list. It will look like Figure 12-6.

Figure 12-6. *Visitor List Page*

We'll be working with two files: the `bloglist.php` file and a corresponding `bloglist.code.php` file in the `includes` directory, which we haven't yet created.

- Create the `bloglist.code.php` file in the `includes` directory.

- Open the `bloglist.php` file and cut the first PHP block from there and paste it into `bloglist.code.php` file (removing the closing PHP tag).

  You should have

  ```
  <?php
  #   bloglist.code.php

      $articles = $paging = $displaying = '';
  ```

- Replace the cut PHP block in the `bloglist.php` file with an include:

  ```
  <?php require_once 'includes/bloglist.code.
  php'; ?>
  ```

We can now fill in the code.

Preparatory Code

Much of what follows can be copied from the `admin-bloglist.code.php` file. We can begin with some preliminary code to set some variables and include some files. We won't need to worry about whether the visitor's logged in, so we won't need the session code:

```
<?php
#   bloglist.code.php

    $root = str_replace($_SERVER['SCRIPT_NAME'], '',
        $_SERVER['SCRIPT_FILENAME']);
    $host = $_SERVER['HTTP_HOST'];
    $protocol = isset($_SERVER['HTTPS']) && $_
    SERVER['HTTPS'] == 'on'
        ? 'https'
        : 'http';
```

```
    $pdo = require_once "$root/includes/db.php";
    require_once "$root/includes/library.php";
    require_once "$root/includes/default-library.php";

    $CONFIG = parse_ini_file("$root/config.ini.php",true);

    $articles = $paging = $displaying = '';
```

Note the reference to the `$CONFIG` array. We'll need that for the number of items for page. We should add that to the `config.ini.php` file:

```
...
[images]
    ...
[gallery]
    page-size = 3
[bloglist]
    page-size = 4
[imagelist]
...
```

We'll also want to get and remember the page number and set the offset and limit for when we fetch the list of articles:

```
<?php
#   bloglist.code.php
    ...

    $articles = $paging = $displaying = '';

    $page = intval($_GET['page'] ?? $_COOKIE['bloglist-page']
    ?? 1)
        ?: 1;

    $limit = $CONFIG['bloglist']['page-size'];
    $offset = ($page - 1) * $limit;
```

```
$blogCount = $pdo -> query('SELECT count(*) FROM blog')
    -> fetchColumn();
$pages = ceil($blogCount / $limit);

$page = max(1, $page);
$page = min($page, $pages);
setcookie('bloglist-page', $page, strtotime('+ 1 hour'));
```

We can now start generating the list of articles.

Generating the Blog Article List

We'll make the article list a little more friendly than the table we used for administration. It will basically be a collection of headings with the article precis and date—something like this:

```
<div>
    <div>
        <h2><a>[Title]</a></h2>
        <p>[Date]</h2>
        <p>[Precis]
    </div>
    <div>
        ...
    </div>
    <div>
        ...
    </div>
</div>
```

Each article preview is wrapped in a `div` element, and the whole collection is wrapped in another `div` element. That's just to make it easier for the layout and the CSS.

First, we can set up some template strings:

```
<?php
#   bloglist.code.php
    ...

    setcookie('bloglist-page', $page, strtotime('+ 1 hour'));

    $h2 = '<h2><a href="blogarticle.php?article=%s">%s</a>
    </h2>';
    $div = '<div>%s<p class="date">%s%s</p>
        <p class="precis">%s</p></div>';
```

The string is in two parts, mainly for convenience. The `$h2` string will be put into the `$div` string.

In the `$h2` string, there's a link to the selected article in the form of `?article=%s`, which is similar to the way we viewed an article previously. The link also has the title of the article.

The `$div` string will have the heading as well as the date and the précis. The date has two `%s` placeholders, one for the created date and one possibly for the updated date.

Next, we'll define the SQL and build an array of articles, which will be imploded at the end:

```
<?php
#   bloglist.code.php
    ...

    setcookie('bloglist-page', $page, strtotime('+ 1 hour'));

    $h2 = '<h2><a href="blogarticle.php?article=%s">%s</a>
    </h2>';
    $div = '<div>%s<p class="date">%s%s</p>
        <p class="precis">%s</p></div>';
```

```
    $sql="SELECT id, title, created, updated, precis FROM blog
        ORDER BY id LIMIT $limit OFFSET $offset";

    $articles=[];
    $articles[] = '<div>';

    foreach($pdo -> query($sql) as [$id, $title, $created,
    $updated,
      $precis]) {

    }

    $articles[] = '</div>';
    $articles=implode($articles);
```

The `$articles` array starts and finishes with `div` tags to contain the entire collection.

As before, we iterate through the query results using the `foreach()` statement, with the columns destructured into the array of variables.

Inside the `foreach()`, we process the dates and then use `sprintf()`, first to generate a heading and then generate the item to add to the `$articles` array:

```
foreach($pdo -> query($sql) as [$id, $title, $created, $updated,
  $precis]) {
    $updated = $updated != $created
        ? sprintf(' (Updated: %s)', date('d M Y g:i a',
            strtotime($updated)))
        : '';
    $created = date('d M Y g:i a',strtotime($created));
    $heading=sprintf($h2, $id, $title);
    $articles[]=sprintf($div, $heading, $created,$updated,
    $precis);
}
```

Note that we've used the same technique on the `$updated` variable as we did for the blog article, to only display it if it's different to the `$created` value.

Finally, we can add the `$paging` and `$displaying` variables for the navigation block:

```
<?php
#   bloglist.code.php
    ...

    $paging = paging($page, $pages);
    $displaying = "Page $page of $pages";
```

When you select the **Oz Blog** link, you'll see a paged list of articles, with a link to each article in the headings.

The jumping kangaroos () is just a CSS trick.

Summary

This has been quite a long chapter, but much of it has been familiar territory, though there have been a few new tricks.

There are many ways we could have worked the code for this final section, but in this chapter, we've tried to develop some solid principles in how to write reliable and maintainable code.

Here are some of the principles:

- Separate the logic from the rest of the code.

  To begin with, we have a code block in a separate file. It doesn't have to be a separate file, but it should be separated from the rest of the web page.

  Next, we developed a configuration section to allow us to set arbitrary values which are not fundamental to the logic.

Finally, we limited the PHP in the HTML part of the page to outputting results and switching between the sections. That leaves the HTML relatively maintenance free while we develop the code.

- Expect changes.

  A number of times, we revisited sections of code to add more features, such as additional attributes or elements in HTML, or options to adjust the way the code works. This is also the case with a number of functions which were revisited to enhance them.

  In all cases, you should try to write your changes in such a way that the original behavior is still supported. For example, when adding parameters to existing functions, we add defaults consistent with the original behavior.

  Of course, it's not always possible, but you should aim to minimize disruption in your code changes.

- Write readable code.

  Coding is hard enough without wading through a swamp of illegible code trying to work out what's going on. Remember that in six months' time you may have to return to the code to make changes or to fix something.

  All of the code samples in this book are scrupulously indented, the variable and function names have been simple and meaningful, comments are used, and even empty lines are used to section code. All developers will have their own preferences as to how this is done, but you should always write your code with these ideas.

- Look for opportunities to reuse your code.

  Obviously, we've made use of copy and paste when writing code similar to other parts of the project. That works better if your code in different sections is organized in similar ways. However, we also made use of functions to write code which can be called from more than one place. Using functions also helps if you need to rewrite some of the behavior.

  Truly generic code can be placed in a separate library file for use in other projects. Less generic code can still be written up in functions for reuse within a script.

  Note that there are some practical limitations in writing functions. Sometimes, we need too many variations, or the code is too trivial, and the use of too many functions can make your code somewhat cryptic.

- Organize your code.

  More easily said than done, of course, but you need to avoid writing great slabs of code which don't follow any clear plan.

  Again, individual developers have personal preferences, but your code organization should include how you name your files, as well as your variables and functions. Develop a pattern for where you put your configuration and initialization code, as well as where you place your support functions and separate blocks of code.

Throughout the project, there are many places where you could have written the code differently. Indeed, some of the code in this chapter was written differently and more tightly than similar code in previous chapters. Now that you presumably know more about writing PHP than in the beginning, you can revisit the previous code and look where you might rewrite it to improve it.

The thing is you need to start developing your own patterns and, wherever possible, to stick to them. This will take time and experience, which is, of course, all part of the fun.

APPENDIX A

Adding Markdown to Your Blog Articles

You might have noticed a distinct lack of styling in your blog articles. All you have is paragraph and line breaks.

You can add some richness to your article text in a number of ways. Traditionally, this was done by adding HTML. However, that's open error and abuse. PHP has a number of functions which can filter or disable HTML, but then it starts to get very complicated. Besides, you still need the person writing the article to know some HTML.

There is a simple alternative language called **markdown**. It was proposed by John Gruber, and you can learn more about it at `https://daringfireball.net/projects/markdown/syntax`. The idea is that a few simple codes would translate into HTML, which simplifies writing a great deal.

Here, we can implement a simplified subset of markdown which is easily added to an article. Since you might not want all of your articles to be in markdown format, we'll make a few changes to the database to store the setting and a number of changes to the rest of the code to implement it.

M. Simon, *An Introduction to PHP*, https://doi.org/10.1007/979-8-8688-0177-8

The Markdown Language

We're going to use a subset of the most useful features of markdown:

- Headings: `## heading`

  ➔ `<h2>heading</h2>`

  Headings start with one or more hashes (#). The number of hashes is the heading level.

  You shouldn't include an `h1` or `h2` because they're supposed to be the main headings of the page. In our simple markdown interpreter, heading levels before 3 are ignored.

- Anchors: `[text](url)`

  ➔ `<a href="url">text</a>`

  In our markdown interpreter, the links include the `target="_blank"` attribute to open them in a new window or tab.

- Strong: `__strong text__`

  ➔ `<strong>strong text</strong>`

- Emphasis: `_emphasised text_`

  ➔ `<em>emphasised text</em>`

- Unordered list: `- item` … etc

  ➔ `<ul><li> ... </li><li> ... </li></ul>`

  One item per line starting with a minus (`-`) and then a space.

- Ordered list: `1. item` ... etc

  ➔ `<ol><li> ... </li><li> ... </li></ol>`

  One item per line starting with a number, a dot, then a space. The actual number doesn't matter—you don't have to write a sequence.

- Block quotes: `> quoted text`

  ➔ `<blockquote>quoted text</blockquote>`

- Line breaks: `text(space)(space)`

  ➔ `text<br>`

  End the text with *two* spaces, and start the next line immediately.

- Paragraphs

  All other text will be interpreted as paragraphs. The text must have double line breaks.

In the `default-library.php` file is a function called `md2html()`. This is *very* heavily inspired by a package called Slimdown (https://github.com/jbroadway/slimdown).

You can see a sample of a markdown text in `setup/bush-hut.md`. It's just a plain text file, but, when you've finished, it will become rich text.

Adding Markdown to the Database

We're going to add another column to the `blog` table to record whether the article uses markdown.

To make changes to a database table, we use the `ALTER TABLE` statement. This can have rather serious consequences if you make a mistake, such as dropping the wrong column. There's no undo in SQL.

Here, we'll add a column called `markdown`.

In PHPMyAdmin, select the `australia` database. In the SQL tab, run the following:

```
ALTER TABLE blog ADD COLUMN markdown BOOLEAN DEFAULT FALSE;
```

This adds another column called `markdown`, which defaults to `false`. If we want it to be `true`, we'll need to set it deliberately.

Adding a Checkbox to the Form

We'll add a checkbox to the `editblog.php` page, similar to the `gallery` checkbox in the `editimage.pph` page:

```
<fieldset id="content" <?= $disabled ?>>
    ...
    <p><label>Article<br>
        <textarea id="article" name="article">
        <?= $article ?></textarea></label>
    </p>
    <p><label>
        <input type="checkbox"
            name="markdown" <?= $markdown ?>> Markdown</label>
    </p>
</fieldset>
```

The additional field includes a variable called `$markdown` which will be `checked` or not. As before, note the two closing angle brackets on the `input`: `<input ... <?= $markdown ?>>`; one closes the PHP output tag and the other closes the HTML element.

In the manage-blog.code.php page, add the $markdown variable to the initialization section:

```
#   manage-blog.code.php
    //  Initialise
        $title = $precis = $article = '';
        $id = 0;
        $errors = '';
        $disabled = '';
       $markdown = '';
```

If you load the editblog.php page, either to add a new article or to edit an existing one, you'll now see the checkbox unchecked.

Reading the Markdown Setting

The getArticle() function reads the article data from the blog table. We'll add the markdown column to what we're fetching:

```
#   manage-blog.code.php

    function getArticle(int $id) {
        global $pdo, $CONFIG, $previewImage;
        $sql = "SELECT title, precis, created, updated,
        article,
            imageid, markdown FROM blog WHERE id=$id";
        ...
        [$title, $precis, $created, $updated, $article,
          $imageid, $markdown] = $row;
        ...
        return [$title, $precis, $created, $updated, $article,
            $imageid, $previewImage, $markdown];
    }
```

We have also copied the data into an additional $marked variable and added that to the array in the return statement.

We should now include the $markdown variable in the prepare update and prepare delete code:

```
if(isset($_POST['prepare-update'])) {
    ...
    [$title, $precis, $created, $updated, $article, $imageid,
        $previewImage, $markdown] = getArticle($id);

    $markdown = $markdown ? ' checked' : '';
    ...
}

if(isset($_POST['prepare-delete'])) {
    ...
    [$title, $precis, $created, $updated, $article, $imageid,
        $previewImage, $markdown] = getArticle($id);

    $markdown = $markdown ? ' checked' : '';
    ...
}
```

We have also converted the value to either checked or an empty string.

At this point, of course, we won't see any changes because none of the current articles use markdown.

Manipulating the Markdown Setting

When an article is inserted or updated, we'll need to interpret and process the new checkbox.

First, we'll read the markdown checkbox with the other values in the getBlogData() function:

```
function getBlogData() {
    ...
    $image = null;
    $markdown = intval(isset($_POST['markdown']));

    ...
    return [$id, $title, $precis, $article,
        $image, $markdown, $errors];
}
```

We also need to return it, just before the $errors value.

The getBlogData() function is called when we insert or update an article. We'll need to add the $markdown variable to these:

```
if(isset($_POST['insert'])) {
    [$id, $title, $precis, $article, $image, $markdown,
$errors]
        = getBlogData();

    ...
}

if(isset($_POST['update'])) {
    [$id, $title, $precis, $article, $image, $markdown,
$errors]
        = getBlogData();

    ...
}
```

We have the addBlogData() function to add the data to the blog table. We'll add the $markdown parameter at the end, making it optional. This value needs to be added into the INSERT statement, both into the column list and the placeholders and into the $data array:

```
function addBlogData(string $title, string $precis,
  string $article, array|string $image=null, string
  $created=null,
  string $updated=null, int $markdown=0) {
    ...
    $sql = 'INSERT INTO blog(title, precis, article,
        created, updated, markdown)
        VALUES(?, ?, ?, ?, ?, ?)';
    $pdoStatement = $pdo -> prepare($sql);
    $data = [$title, $precis, $article, $created?:
    date('Y-m-d H:i:s'),
        $updated?:date('Y-m-d H:i:s'), $markdown];
    ...
}
```

The addBlogData() function is called from two places. First, it's called when we add an individual article. We'll need to read in the $markdown value:

```
   if(isset($_POST['insert'])) {

    [$id, $title, $precis, $article, $image, $markdown,
    $errors]
        = getBlogData();

    if(!$errors) {  //  proceed
        $id = addBlogData($title, $precis, $article, $image,
            null, null, $markdown);
        ...
    }
```

```
    else {          //  handle error
        ...
    }
    ...
}
```

Second, it's called when we import data. That's a little problem here in that the code originally didn't allow for the extra data, so the CSV file may still be the older style without it. We'll need to allow for that.

The first thing is to add the extra column to the `$keys` array string:

```
$keys
 = explode(' ','title precis article created updated image
 markdown');
```

If the imported row doesn't have the additional column, we can pad it with a `null` before combining it with the keys:

```
foreach($data as $row) {
    $row = array_pad($row, 7, null);
    $row = array_combine($keys, $row);
    ...
)
```

The `array_pad()` function takes the original array and pads it to a number of values, in this case to `7` values with an extra `null`. If the array is already the right size, the padding is ignored.

Finally, you can include it in the call to `addBlogData()`. You don't have to if you're using the `...` operator, but you will if you're adding the parameters individually:

```
//  addBlogData($row['title'], $row['precis'], $row['article'],
//    $row['created'], $row['updated'],
//    $row['image'], $row['markdown']);  //  Any PHP
```

This gives us something like the following:

```
if(isset($_POST['import'])) {
  ...
  if(!$errors) {
    ...
    $keys = explode(' ',
        'title precis article created updated image markdown');

    foreach($data as $row) {
        $row = array_combine($keys, $row);
        $row = array_pad($row, 7, null);
        ...
            addBlogData(...$row);                  //  PHP >= 8
            //  addBlogData(...array_values($row));
                // PHP >= 7.2
            //  addBlogData($row['title'], $row['precis'],
            //    $row[article], $row[created], $row[updated],
            //    $row[image], $row['markdown');   //  Any PHP
    }
    ...
  }
  else {
     ...
  }
}
```

We'll also need to include the markdown value when updating the article. We've already read the data, so it's just a matter of including it in the SQL statement and the data:

```
if(isset($_POST['update'])) {
    [$id, $title, $precis, $article, $image, $markdown,
    $errors]
```

```
        = getBlogData();
    ...
    if(!$errors) {  //  proceed
        $sql = 'UPDATE blog SET title=?, precis=?, article=?,
            updated=?, markdown=? WHERE id=?';
        $pdoStatement = $pdo -> prepare($sql);
        $data = [$title, $precis, $article, date
        ('Y-m-d H:i:s'),
            $markdown, $id];
        ...
    }
```

We now have the markdown flag added to the database and the data.

You might try adding a new article with markdown data before the final step. You can copy the markdown from the `setup/bush-hut.md` file and add the image `setup/bush-hut.jpg` or whatever takes your fancy.

Displaying the Markdown Article

Now that we have the markdown column in the data, we'll need to read it when we're displaying the article.

In the `blogarticle.php` file, we'll add a variable to the `article` element:

```
<article id="blogarticle" <?=$markdown?>>
    ...
</article>
```

Watch out for the two closing angle brackets (`>>`) and the extra space before the PHP output.

This variable may contain a `class` attribute of `class="markdown"`. That's so that CSS can control its appearance.

In the blog.code.php file, add the variable to the initialization:

```
$title = $precis = $created = $updated = $article
    = $figure = $markdown = '';
```

Then add the column to the SQL statement and to the list of variables:

```
<?php
#   blog.code.php
    ...
    $sql = "SELECT title, precis, created, updated, article,
        imageid, markdown
        FROM blog WHERE id=$id";
    $row = $pdo -> query($sql) -> fetch();

    ...

    [$pagetitle, $precis, $created, $updated, $article,
        $imageid, $markdown]  = $row;

    ...
```

Toward the end of the code, we run the article through the pilcrow2p function to convert the stored pilcrows to paragraphs. We'll change that to test for the $markdown variable:

```
$updated = $updated != $created
    ? sprintf(' Updated: %s',
        date('d M Y g:i a', strtotime($updated)))
    : '';
$created = date('d M Y g:i a', strtotime($created));

if(!$markdown)  $article = pilcrow2p($article, true);
else {
    //  markdown
}
```

If the $markdown variable is set, we'll convert the pilcrows to real line breaks and then convert the article itself:

```
if(!$markdown)  $article = pilcrow2p($article, true);
else {
    $article = pilcrow2nl($article);
    $article = md2html($article);
    $markdown = ' class="markdown"';
}
```

We've also set the $markdown variable to add the class to the article. You should now see something like Figure A-1.

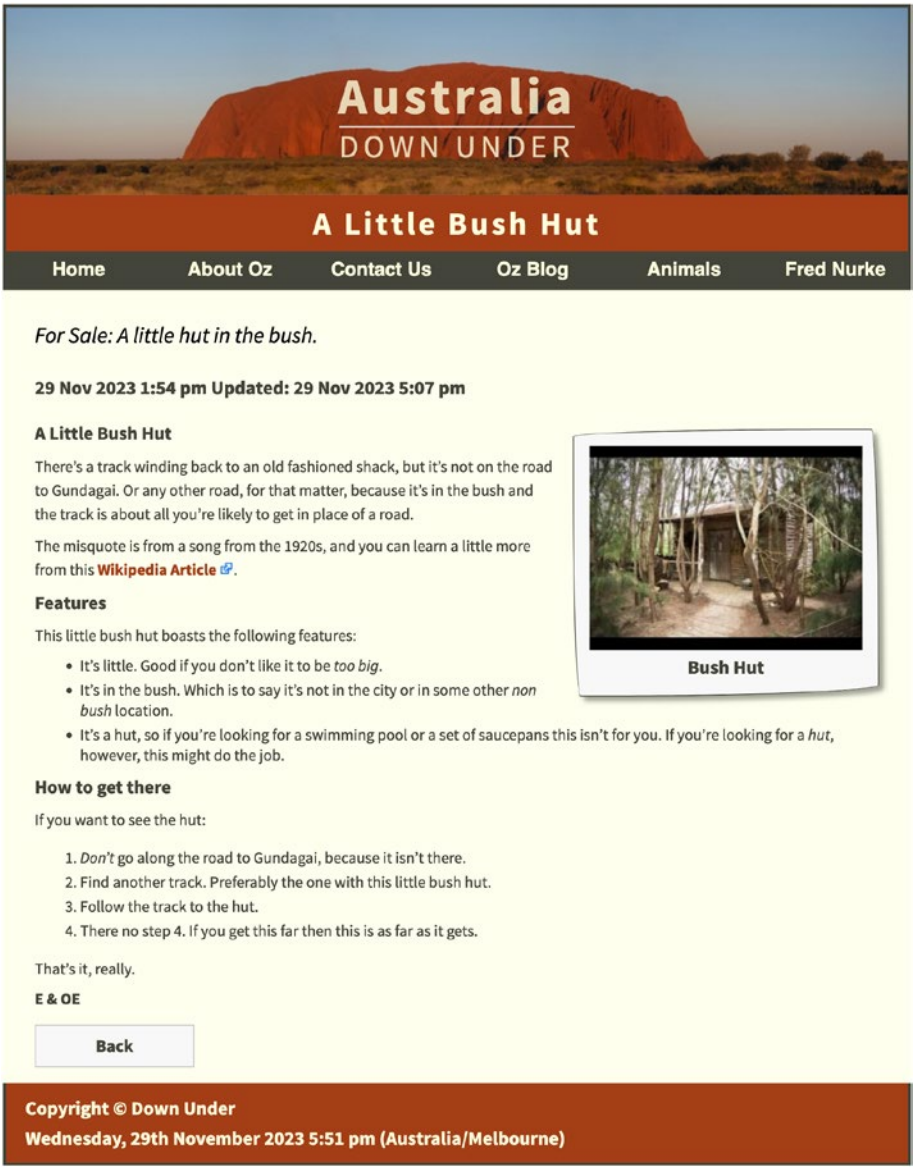

Figure A-1. *Markdown Blog Article*

APPENDIX B

Non-PHP Tricks

Not all of the magic on the sample project comes from PHP. Obviously, there's some HTML and SQL involved, but some of the special tricks are achieved through some extra CSS and JavaScript.

If you know nothing of JavaScript or CSS, and care even less, then you can ignore the rest of this appendix. If, on the other hand, you know something about them, then you can learn a little more of how these techniques are used in the project.

Toggle Background

This isn't so much a special effect as simply an aid to troubleshooting.

You'll notice that the background is *very* busy with a vibrant repeated image. That's fine until you try to read some of the error messages or the output from troubleshooting print statements.

If you shift-click the banner image, the background will toggle off and on. That's just some JavaScript to change the state of the background and to remember this state.

The code is

```
//  Background Image
    if(localStorage.getItem('debug'))
        document.body.classList.add('debug');
```


M. Simon, *An Introduction to PHP*, https://doi.org/10.1007/979-8-8688-0177-8

```
document.querySelector('header').onclick = event => {
    if(event.shiftKey) document.body.classList.
    toggle('debug');
    if(document.body.classList.contains('debug'))
        localStorage.setItem('debug',true);
    else localStorage.removeItem('debug');
};
```

There is a CSS class called `debug`. If the body has this class, the background image is hidden.

The code searches for the `header` element (`document.querySelector('header')`) and assigns a `click` event listener.

When you click the header element, the event listener checks whether the shift key is down. If so, the `debug` class is toggled.

If the toggle has set the class, the value is stored in the local storage. Otherwise, the value is removed. The local storage is maintained by the browser for each website and is useful for storing setting information.

In the initialization code, JavaScript checks the local storage for whether the debug value has been set before. It sets the class accordingly.

Visible Passwords

By default, password fields on a web form are obscured—all you see when you type is bullets instead of the characters. That's to combat so-called "shoulder-surfing"—someone looking over your shoulder as you type your password.

That's OK if you're logging in in a public place, but often you're at home or in some other private location, and all you get is the downside of obscuring your password: you can't see what you're typing. That's especially annoying if you've followed the best advice and generated a very complex password.

The login form on the sample project has a solution to that which allows you to see the password for ten seconds. It comes in three parts.

First, the HTML has an additional button beside the password field:

```
<label>Password:<br>
    <input type="password" name="password" id="password">
    <button type="button" name="show-password" title="Show
    Password">👀</button>
</label>
```

Of course, by itself it does nothing. It relies on JavaScript to do its work.

The second part is the JavaScript. First, we find the login form and attach an event listener to the `show-password` button. The event listener is the `doShowPassword()` function:

```
let form;
if(form = document.querySelector('form#login')) {
    form.elements['show-password'].onclick
        = doShowPassword.bind(null, form);
}
```

The `doShowPassword()` function is defined separately:

```
function doShowPassword(form, event) {
    form.elements['password'].type = 'text';
    form.elements['show-password'].setAttribute('on', true);
    window.setTimeout(() => {
        form.elements['password'].type = 'password';
        form.elements['show-password'].removeAttribute('on');
    }, 10000);
    event.preventDefault();
}
```

In the `doShowPassword()` function, all that's really necessary is that the password field has been changed to a `text` type, which will readily display its contents. The rest of the function sets up an attribute for CSS and reverts the field type after ten seconds.

The function adds an `on` attribute to the show password button. That's not a standard attribute, but neither JavaScript nor CSS cares about that. What matters is that CSS can target that attribute.

The `setTimeout()` function waits for 10,000 milliseconds and then restores the original state by setting the field type back to `password` and removing the `on` attribute.

The third part is the CSS, and it's only there for show. Most of the CSS for the button is to make the silly eyes look right and place the button at the end of the password field.

However, there are two important properties to give the visual effect:

```
form#login button[name="show-password"] {
    ...
    transform: scale(-1, 1);
    transition: transform 0.75s;
}
```

The `transform` property is technically a `scale` effect, but the first (horizontal) value of `-1` effectively flips the image horizontally. Its natural appearance is to look stage right.

The `transition` property slows down the change to 0.75 of a second.

When the `on` attribute has been set, the following CSS is applied:

```
form#login button[name="show-password"][on] {
    transform: scale(1.6);
}
```

Both horizontal and vertical axes are scaled to 1.6 (enlarged), and the positive value flips the appearance to stage left.

Required Upload

In the `uploadimage.php/editimage.php` file, there's a form to upload a ZIP file. You'll have noticed that the submit button doesn't appear until a file has been selected.

This effect comes in two parts: the HTML which comprises the form and the CSS which toggles the submit button.

The important part of the HTML looks something like this:

```
<form id="editimage-import" ...>
    <input type="file" name="import-file" required>
    <button type="submit" name="import">Import</button>
</form>
```

The `file` input has the `required` attribute, which is an HTML5 validation attribute. Once all the form's validation requirements have been satisfied, the form is **valid**. Until then, the form is **invalid**. In this case, there's only one requirement, which is that a file must be selected.

The form also has an `id`. You don't technically need that for a form, but it's useful if you have some additional work to do in JavaScript, or you want to do something in CSS, such as hiding the button.

The relevant part of the CSS is

```
form#editimage-import input[name="import-file"]:invalid + button,
form#editblog-import input[name="import-file"]:invalid + button {
    display: none;
}
```

This selects for a button inside the form. There are three special parts of this selector:

- The `[name="import"]` is called an **attribute selector** and specifies an element with the `name="import"` attribute. It's not strictly necessary here as there's only one button in the form, but it's always best to be specific.
- The `:invalid` is a **pseudoclass selector**. In CSS, you use it like a class, but its value is set automatically by the state. In this case, an incomplete form would trigger the `invalid` state.
- The + combinator is called a **sibling** combinator and indicates a `button` which is adjacent but *after* the invalid input.

As you see, while the form is invalid, the button is hidden using the `display: none` property.

Preview Image

There are a number of places where an image is previewed. This is all done with JavaScript.

Previewing a Referenced Image

For the blog management, you have the option of choosing a selected image, either with a select menu or a radio button. In both cases, the image options are wrapped in an anchor which references a larger image.

In the JavaScript, the initialization script looks for a `div` element whose `id` is `use-image`:

```
if(useImage = document.querySelector('div#use-image')) {
    let previewImage
        = document.querySelector(img#preview-new-image);
    useImage.onclick = event => {
        showImage(
            previewImage, event.target.dataset.preview,
            {dw: 240, dh: 180}
        );
    };
}
```

If it finds one, it then looks for an `img` element called `preview-new-image`.

Finally, the `div` element gets a `click` event listener, which calls the `showImage()` function, with the `img` element to be populated; the `data-preview` attribute, which is a reference to a larger image; and an object with the maximum size of the preview. In JavaScript, this becomes the `dataset.preview` property.

The `showImage()` function populates the preview image with the new image:

```
function showImage(preview, src, limit) {
    preview.src = '';
    [preview.width, preview.height] = [limit.dw, limit.dh];
    let cache = new Image();
    cache.src = src;
    cache.onload = e => {
        let img = e.target;
        if(limit) {
            if(img.width/img.height < limit.dw/limit.dh)
                preview.width = limit.dh * img.width /
                img.height;
```

```
            else preview.height = limit.dw * img.height /
            img.width;
        }
        preview.src = img.src;
    };
}
```

If it were not for the need to adjust the size, the function would be much simpler. The new image needs to be scaled to fit within this limit, which is where you see the `preview.width` and the `preview.height` values being adjusted.

The problem is that you can't do anything with the image size until after it's finished loading. The `cache` variable is used to hold a copy of the image. When it's finished loading, it will fire the `load` event. Inside that event listener, the size adjustments are made, and the image is copied into the visible preview.

The File Input

The file input buttons are expected to be used to attach an image. The initialization script looks for file input buttons with the `data-preview` attribute:

```
if(elements
  = document.querySelectorAll('input[type="file"][data-
  preview]')) {
    elements.forEach(element => {
        doFilePreview(
            element,
            document.querySelector(img#${element.dataset.
            preview}),
            {dw: 240, dh: 180}
```

```
        );
    });
}
```

On any one page, there's only one, which will be processed by the `doFilePreview()` function.

The `doFilePreview()` is responsible for remembering the original preview image, previewing the selected image, and for restoring the original:

```
function doFilePreview(fileInput, preview, limit) {
    if(!preview) return;
    //  Remember Original
        preview.original = {
            src: preview.src,
            width: preview.width,
            height: preview.height
        };
    //  Shift-Click -> Reset to Original
        fileInput.onclick = event => {
            if(event.shiftKey) {
                fileInput.value = null;
                event.preventDefault();
                preview.src = preview.original.src;
                preview.width = preview.original.width;
                preview.height = preview.original.height;
            }
        };
    //  Preview Attached File
        fileInput.onchange = () => {
            try {
                let reader = new FileReader();
                reader.readAsDataURL(fileInput.files[0]);
```

```
                if(limit)
                    [preview.width, preview.height]
                    = [limit.dw, limit.dh];
                reader.onload = event => {
                    showImage(preview, reader.result, limit);
                };
            }
            catch (error) {
                preview.src = '';
            }
        };
}
```

The `Remember Original` code simply copies the `src` and dimensions of the original image into a new property called `original`.

The `Reset to Original` code listens for a `click` event and tests for the shift key. If this happens, the `src` and dimension properties are copied from the `original` property back to the preview image. This isn't the normal behavior of a file input, of course, so the `preventDefault()` method is called to stop the button from attempting to upload another file.

The `Preview Attached File` code listens for a change to the file input, which is what happens after you select a file. It then uses a `FileReader` object to read contents of the file and generates a binary version using the `data://` pseudo protocol. From there, it is sent to the `showImage()` function to be displayed.

Light Box

The pop-up image you see, especially if you click an icon in the image management, is referred to as a **light box**. The actual code is in the `lightbox.js` file and is much too involved to be discussed here.

The initialization script specifically looks for a container element to be used in the light box:

```
if(document.querySelector(container = 'table.manage'))
    doLightbox(container);
if(document.querySelector(container
  = 'article#blogarticle>div#article>figure'))
    doLightbox(container);
```

The light box code handles the following:

- Creates the elements for the light box: a pop-up with the image and a caption and the background to cover the screen.
- Creates the CSS to manage the appearance and disappearance of the pop-up.
- Attaches an event listener to the anchor around the image, which will show the referenced image in the pop-up.
- Attaches an event listener to the background to hide the image, as well as a key event listener for the escape key to do the same thing.

The actual zooming effect as well as the shape of the pop-up is handled in the CSS and can be adjusted in the `lightbox.css` file.

Paging Page Number

When you hover over a paging button, the page number appears in its place. This is pure CSS, the relevant part of which is the following:

```
p#paging a {
    position: relative;
}
```

```
p#paging a:hover:before {
    content: attr(data-page);
    position: absolute;
}
```

The rest of the CSS is there to make it look right.

The anchor element has its `display` property set to `relative`. That won't change the appearance of the anchor itself, but is to prepare for the next part.

The `a:hover:before` selector applies only when the mouse is hovered over the anchor. The `:before` is a pseudo element which refers to the content at the beginning of the element, which is normally empty. In this case, it's to be filled with the `data-page` attribute, which, as you will recall, was added to each of the paging buttons.

The `display` property `absolute` causes the new content to sit in the same position as the original content.

The Hopping Kangaroo

The hopping kangaroo is a pseudo element added to the end of the `h2` element in the visitor's blog list. The relevant part of the CSS is

```
article#bloglist>div>div>h2:after {
    content: "🦘";       /* aka \1F998;*/
    display: inline-block;
    transition: transform 1s;
    position: relative;
}
article#bloglist>div>div:hover>h2:after {
    transform: scale(-1,1);
    animation: hop .5s infinite alternate;
}
```

```
@keyframes hop {
    from {
        top: 0;
    }
    to {
        top: -.5em;
    }
}
```

The `:after` pseudo element represents the end of the content of the `h2` element. In this case, it's filled with the kangaroo icon  . The CSS Unicode for that character is `\1F998`.

The `:after` pseudo element also sets a `transition` property. A transition is a simplified one-off animation. CSS won't transition inline elements, so the `display` property is set to `inline-block`. By itself, the `transition` property does nothing. However, when the listed property `transform` is changed, the `transition` will cause it to change slowly, over a time of one second.

When you hover over the `div` (`div:hover`), it will change both `transform` and `animation` properties.

The transform property is set to `scale(-1,1)`, which reverses the direction without actually resizing the image.

Meanwhile, an animation called `hop` is triggered to run infinitely (or at least while the mouse is over it).

The `position` of the `:after` pseudo element is set to `relative` to allow adjustments. The `hop` animation is contained in `@keyframes` rule. In this case, it simply adjusts the `top` property, relative to the original position.

APPENDIX C

PHP Versions

PHP is, at the time of writing, at version 8 point something. All versions before version 8 are regarded as legacy versions, and you should update PHP as soon as possible.

However, that's not always readily done. Some organizations are skeptical of newer releases, some hosts are very conservative, and some third-party packages use legacy code which fails in new versions of PHP.

To be clear, none of this should be a problem. New versions of PHP sometimes do have breaking changes, but these changes have gone through a number of stages. First, old features are *deprecated*, which means they're discouraged and marked for removal. Then they start to issue warnings. Finally, the old features are removed altogether. Web administrators should try to keep up with this.

If you're working with an older version of PHP, some of the code in this book may need a few minor changes. For the most part, new versions have added features which are more convenient to use. This appendix lists the most relevant changes.

The array() and list() Language Constructs

These have been around forever.

Although these look like functions, they're not. They are called **language constructs**. You use them like functions, but on the inside they're handled differently. There are also some things you can't do with them that you can with real functions.

M. Simon, *An Introduction to PHP*, https://doi.org/10.1007/979-8-8688-0177-8

The `array()` expression generates an array:

```
$data = array('apple', 'banana', 'cherry');
```

As of **PHP 5.4**, it's easier to use the shorter syntax:

```
$data = ['apple', 'banana', 'cherry'];
```

The `list()` expression assigns multiple variables from an array. For example:

```
$data = ['apple', 'banana', 'cherry'];

list($a, $b, $c) = $data;

//  equivalent to:
    $a = $data[0];
    $b = $data[1];
    $c = $data[2];
```

As of **PHP 7.1,** you can use the **array destructuring** syntax:

```
$data = ['apple', 'banana', 'cherry'];

[$a, $b, $c] = $data;
```

Before **PHP 7.1**, you were limited to numeric keys, and you didn't specify them. With current versions, you can name your keys:

```
$data = ['apple', 'banana', 'cherry'];
[0 => $a, 1 => $b, 2 => $c] = $data;

$data = ['first' => 'apple', 'second' => 'banana',
    'third' => 'cherry'];
['first' => $a, 'second' => $b, 'third' => $c] = $data;
```

This is also true if you use the classic `list()` syntax. When using keys, you specify the key first and then the variable.

Arrays can have mixed numeric and string keys, such as the array you get from `getimagesize()`. You can also use mixed keys in `list()` or array destructuring, but you must specify all of the keys or none of them. If you specify none of them, then only the values with numeric keys will be assigned.

The … Operator

That's what PHP calls it, which doesn't exactly roll off the tongue. If you look hard enough, you might see it called the **ellipsis** operator. In other languages, it's sometimes called the **spread** or **splat** operator. It would possibly be more helpful to call it the **packing/unpacking** operator, because that's what it does.

Introduced in **PHP 5.6**, it's used to pack or unpack an array.

If you use the operator in a function definition, it packs multiple arguments into a single array. For example:

```
function doit($thing, ...$etc) {
    //  whatever
}

doit('something', 'something else', 'stuff', 'more stuff');
```

Here, the first value is sent to the `$thing` parameter, while the other three values are packed into the `$etc` array.

This type of function is called a **variadic** function as it takes a variable number of parameters. Traditionally, especially before the `...` operator became available, you would call `func_get_args()` to get all of the arguments in an array.

If you use the operator when calling a function, it unpacks an array into individual parameters. For example:

```
function doit($item, $price, $quantity) {
    //  whatever
}

$data = ['book', 23, 4];
doit(...$data);
```

Here, the `$data` array is unpacked into individual values to be sent to the function.

As of **PHP 8**, you can also unpack an associative array. However, the keys need to match the parameter names.

mail() Function Additional Headers

The `mail()` function includes an argument for the additional headers, which should always include the `From:` header.

You can set the additional headers in a string:

```
$to = '...';
$subject = '...';
$message = '...';

$headers = "Date: ...\r\nFrom: ...\r\nCc: ...";

mail($to, $subject, $message, $headers);
```

The Email Standard requires the **CRLF** line break between headers, so this is encoded in `\r\n` as before. This requires a double-quoted string to work—single quotes don't support most special character codes.

You also need to have a space after the colon, but not before the next header, since a space at the beginning of a line indicates continuation.

As of PHP 7.2, these headings can be in an array:

```
$to = '...';
$subject = '...';
$message = '...';

$headers = [
    'Date' => '...',
    'From' => 'me@example.net',
    'Cc' => 'fred@example.com'
];

mail($to, $subject, $message, $headers);
```

The `mail()` function takes care of the rest.

If you want to use an array in an older version of PHP, you'll need to convert it to a string. Something like the following will do:

```
$headers = [
    'Date' => '...',
    'From' => 'me@example.net',
    'Cc' => 'fred@example.com'
];

$headers = array_map(
    function($k, $v) {
        return "$k: $v";
    } , array_keys($headers), array_values($headers)
);
$headers = implode("\r\n", $headers);
```

The `array_map()` function will take the keys and values of the `$headers` array and return each item of the `$headers` array as a string of the form `header: value`. The resulting array is then imploded into a single string.

Function Data Types

Some programming languages associate variable names with data types. For example, you can declare a variable to be a string or an integer. PHP doesn't do that and probably never will—any variable can be assigned a value of any type at any time.

However, PHP does allow you to specify data types when defining a function. For example:

```
function process(int $first, int $second): string {
    $first = 'hello';
    return "Got $first and $second";
}
print process('3', 5.7);
```

This requires that the inputs be *compatible* with the particular data types. In this case, they must be compatible with integers. Note that in the example, neither of the two values sent to the function is an integer. The first is a string which can be converted to an integer, while the second is a decimal which will be truncated to an integer.

What happens inside the function is another matter: there's nothing preventing the code from changing the variable to something else, such as a string.

The return value, however, is guaranteed to be a string.

The purpose of type hinting is to provide some safety to function calls. If the inputs are guaranteed to be the correct types, then it's easy to guarantee the output.

You don't need to specify the data types for all of the parameters.

The list of acceptable data types has been growing over the years. Here are some of the changes:

- **PHP 7.1**
  - The data can be nullable, such as `?int $a`.
  - The return type can be `void` (no return value).
- **PHP 7.2**
  - The type can be an object type.
- **PHP 8.0**
  - You can have **union** types: that is, you can specify alternative types.

There is more information at www.php.net/manual/en/language.types.declarations.php.

Named Parameters

PHP function parameters have a number of features which help to make functions more flexible:

- Parameters can be type hinted (discussed in the previous section).
- You can have optional parameters, by setting default values.
- You can use the `...` to pack multiple arguments into an array (discussed in a previous section).

As of **PHP 8.0**, you can now use **named parameters**. For example:

```
function doit($a, $b, $c) {
    print $a / $b + $c;
}
```

```
doit(b: 3, a: 21, c: 20);   //  21 / 3 + 20 = 27
doit(24, 6, 10);            //  24 / 6 + 10 = 14
doit(21, c:20, b: 7);       //  21 / 7 + 20 = 23
```

This gives you a number of options:

- You specify the value of the argument after its parameter name, *without* the dollar sign ($). This can be in any order.
- You can ignore the names and specify the argument values in the traditional way. Of course, you need to get them in the correct order. If you do, they are referred to as **positional** arguments.
- You can mix positional arguments with named arguments. However, you need to dispose of the positional arguments first.

This is particularly useful when you have a function with a large number of optional parameters, and you don't want to specify them all:

```
function doit($a=12, $b=3, $c=20) {
    print $a / $b + $c;
    print "\t//\t$a / $b + $c\n";
}

doit(b: 4);             //  12 / 4 + 20 = 23
```

Even built-in PHP functions, such as the `imagecopyresampled()` function, which has *ten* parameters in a mysterious order, allow you to use parameter names to make them more readable.

APPENDIX D

Default Library Functions

In the book, we've tried to concentrate on developing the core skills which you're likely to find most useful. Some of that involved using code which is useful, but on a tangent to what we're focusing on.

In this appendix, we look at some of the functions which took on some of that extra work. By now, they're mostly within your understanding, though you may have to take a little on faith.

The functions we look at are

- `resizeImage()`: The function which copies and resizes images for the image gallery and the blog. This is covered in detail, including the mathematics you need to get the scaling right.

  It also includes the `splitSize()` function which splits a dimension string into its parts.

- `unzip()`: A function which unzips a ZIP file into a directory. This is already discussed in the book, but here we add a little more detail.

- `md2html()`: A very light-duty markdown interpreter, used in Appendix A. We don't go into complete detail but discuss the use of regular expressions in interpreting the code.

M. Simon, *An Introduction to PHP*, https://doi.org/10.1007/979-8-8688-0177-8

- Two minor functions which are simply wrappers for other functions. The `MimeType()` function is used to discover the MIME type of a file and is a convenient wrapper around a more obscure technique. The `printr()` function is a wrapper to make the output from the `print_r()` built-in function more readable.

One thing you'll learn is some more techniques, which is always a good thing.

The other thing you'll learn is that sometimes it's OK to lean on somebody else's work. You'll see that, for example, in the discussion on the `md2html()` function which, in turn, borrows from another party.

Resizing Images

In the sample project, you used a provided function to generate thumbnail copies of uploaded images. Here, we're going to have a look at how that function is developed.

When developing a function, there's often the question of how best to implement the input parameters. You don't want too many (you'll see an example of a PHP function with too many parameters later), but you'll want some flexibility. You may decide that the way it's done in this function could have been improved upon, and you're welcome to do so.

You can use PHP to create and manipulate images and files. This ability is not directly built into PHP itself, but is available through the GD library, one of the standard libraries which is normally part of the PHP implementation.

GD originally stood for "GIF Draw," but now informally means "Graphics Draw" (GIF support was actually dropped at one stage due to licensing reasons but has since been re-implemented after the patent expired). Officially, it's just GD.

GD has many functions for loading and saving images, transforming them, simple drawing, and writing text. Here, we won't be drawing anything, though it is possible to use GD functions, say, to superimpose a timestamp as a watermark.

Make sure that you've enabled the GD library, as described in the setup for this book.

Resizing an image in PHP is a matter of copying a larger image into a smaller copy and saving it. However, if the required copy is of a different shape from the original (i.e., the **aspect**, or the ratio of the width to the height, differs), then you will need extra work to pad the difference.

The steps required to resize images are

- Load the original.
- Later, you will need to make padding adjustments for different aspect ratios.
- Create a smaller blank image.
- Copy the original into the smaller image.
- Save the new image.

When you copy the original image into the smaller image, you'll notice that it is probably distorted. This is because the original image may not be the same shape as the copy. This will be fixed with some padding adjustments.

The resizeImage() Function

We'll start a basic function to load an image and resize it into a new image:

```
function resizeImage($source, $destination, $size='160x120',
    $options=[]) {
        //  Set up
```

```
        //  Load Image
        //  Adjust Scale & Dimensions
        //  Create Blank Image
        //  Copy Original into New Image
        //  Save New Image
}
```

This function will take a source file name and save a copy to a destination file. The optional size parameter will be a string, defaulting to `'160x120'`. We've also included an additional `$options` parameter which is an array of whatever we think might be useful options.

Interpreting the Resize Dimensions

After the source and destination parameters, the next obvious parameter would be the output size. We could have used two parameter variables, such as `$width` and `$height`, and you may decide to do just that. In this case, we'll use a string of the form `width x height`, and we'll split it into the width and height.

For testing purposes, we can use a default string of `'160x120'`. You can use `explode()` to split a string. This function allows you to specify a separator string and will return a number array of values.

You can then use array destructuring to copy the values into two variables:

```
»   $size = '160x120';
»   [$width, $height] = explode('x', $size);
```

To be more flexible, you should allow for extra spaces in the string. The `explode()` function isn't flexible enough for this, but you can use the `preg_match()` function, which takes a regular expression and returns component parts.

> If you're trying to pronounce the function name, you might try "p-reg-match." It means something like "Perl-compatible regular expression match." Perl is another programming language which features quite sophisticated regular expressions.

The regular expression we'll use is the following:

```
/^\s*(\d+)\s*[xX]\s*(\d+)\s*$/
```

If you think that's a little cryptic, you may be right. Like all regular expressions, it's pretty unreadable at a glance, but if you break it down, it makes more sense. Here's how this one works:

Code	Description
`/ ... /`	The whole expression is contained in the `/ ... /` delimiters
`/^ ... $/`	The special characters ^ and $ represent the beginning and the end of the string. Basically, the expression needs to match the whole string
`\s*`	The code `\s` represents any spacing character (such as the space or a tab), and the * counts zero or more occurrences. Basically, we're searching for any number of possible spaces
`(...)`	Anything in parentheses will be stored for later. Anything outside parentheses must match, but will then be forgotten
`\d+`	The code `\d` represents any digit (`0 - 9`), while the + counts one or more occurrences. Here, we're looking for at least one digit. Being in parentheses, the result will be stored for later
`[xX]`	This represents a choice of any character in the square brackets. We're looking to match an upper or lower case `x`

To put this in English, we're trying to match some possible spaces, some digits, some more possible spaces, an upper or lower case x, some more possible spaces, some more digits, and some more possible spaces.

The preg_match() function takes the following form:

```
$success = preg_match($pattern, $string, $matches);
```

The function works like this:

- $pattern is the regular expression you'll use to test the string.
- $string is the string to be tested.
- If the string tests successfully, the return value is 1, which is stored in the $success variable. Otherwise, it's 0.
- If the string tests successfully, the matches are stored in an array in the $matches variable. Otherwise, the array is empty.

In this case, a successful array will contain the following items:

Index	Value
0	The whole string
1	The first number
2	The second number

We can store the pattern in a variable for easy maintenance. We'll use the return value in an `if()` statement to decide whether the match was successful. We can then write the code as follows:

```
function resizeImage($source, $destination, $size='160x120') {
    //  Set up
        $pattern = '/^\s*(\d+)\s*[xX]\s*(\d+)\s*$/';
        if(preg_match($pattern, $size, $matches)) {
            //  success
        }
        else {
            //  failure
        }
}
```

If the match is unsuccessful, we can choose to give up immediately with a `return` statement. Alternatively, we can just use some default values, which is more friendly:

```
function resizeImage($source, $destination, $size='160x120',
  $options=[]) {
    //  Set up
        $pattern = '/^\s*(\d+)\s*[xX]\s*(\d+)\s*$/';
        if(preg_match($pattern, $size, $matches)) {
            //  success
        }
        else {
            [$width, $height] = [160, 120];
        }
}
```

If the match is successful, we can destructure the `$matches` array:

```
function resizeImage($source, $destination, $size='160x120',
  $options=[]) {
    //  Set up
        $pattern = '/^\s*(\d+)\s*[xX]\s*(\d+)\s*$/';
        if(preg_match($pattern, $size, $matches)) {
            [, $width, $height] = $matches;
        }
        else {
            [$width, $height] = [160, 120];
        }
}
```

The leading comma on the left of the expression is to skip over the first value which is the entire string.

Loading the Original Image

Now that we've worked out the dimensions, we need to load the image to be resized.

Images stored on the disk, or transmitted over the Internet, are usually stored in a special format which includes compressing them to take less space, as well as other data about the image. These formats include GIF, JPEG, and PNG, which are the classic browser image formats. We'll include support for these formats in this function.

On the other hand, images in memory are uncompressed and generic. This means that you can load from one format on the disk and later save the image in another format.

While PHP has functions to do this, they are specific to individual formats. You will have to choose which of many similar functions to use to load or save images, depending on the original input format and the desired output format. There are also functions which can discover data from the images and, of course, to make changes.

Modern browsers support many other newer formats, such as APNG, WEBP, and AVIF, but not all to the same extent. PHP also supports these newer formats, though you'll have to make do with the PNG functions for the APNG format.

We won't support these formats here, but you can add support using the functions available at `www.php.net/manual/en/ref.image.php`.

Getting Information About the Original Image

To begin with, you will need to read the original image into memory. Unfortunately, GD has not one but many functions to load an image into memory, depending on the image file type. This will also involve working out the format of the original image, as well as other details.

Before loading an image, you need to know its file type. PHP has a function inaptly called `getimagesize()` which returns information about the type, as well as the image's physical dimensions.

The return from a call to `getimagesize()` is an array of values with the width, height, and type and a string with the dimensions for an HTML img tag, as well as other prices of data. For example, given a JPEG file whose dimensions are `640 x 480`:

Key	Sample Value	Meaning
`0`	`640`	Width
`1`	`480`	Height
`2`	`2`	Image type constant
`3`	`width="640" height="480"`	Dimensions for `img` tag
`...`		
`mime`	`image/jpeg`	MIME type

Note that the image type constant will in fact be an integer (in this case, `2` is a JPEG file). PHP has a large number of built-in constants, including names for the image types, but the MIME type is preferable because it is more meaningful.

The dimension string is purely a convenience for inclusion in an `img` element.

Since the data comes as an array, you could read it as follows:

```
$imageInfo = getimagesize($source);

$sw = $imageInfo[0];
$sh = $imageInfo[1];
$mime = $imageInfo['mime'];
```

We're using `$sw` and `$sh` for the source image dimensions ("source width" and "source height") to distinguish these from the `$width` and `$height` parameter variables. We won't need the rest of the data from the `getimagesize()` function.

More practically, you could read the first two values into variables by using simple array destructuring; however, you would still need to read the nonnumeric key separately, as the simple format only supports numeric keys:

```
$imageInfo = getimagesize($source);
[$sw, $sh] = $imageInfo;
$mime = $imageInfo['mime'];
```

Even more practically, you can, if you want, include the string key using the extended array destructuring format:

```
[0=>$sw, 1=>$sh, 'mime'=>$mime] = getimagesize($source);
```

Here, we've dispensed with the `imageInfo` variable and done it all in one step.

We can now add it to our function:

```
function resizeImage($source, $destination, $size='160x120',
  $options=[]) {
    //  Set up
        ...
    //  Load Image
        [0=>$sw, 1=>$sh, 'mime'=>$mime] =
        getimagesize($source);
}
```

Now that we know the image type, we can load it into memory.

Loading the Image Data

Loading an image data is a matter of a single function call, but choosing which function to call is more involved. This will depend on the image type, which is one of the pieces of information previously gathered. You can then use a switch statement to choose the appropriate function:

```
//  Load Image
    [0=>$sw, 1=>$sh, 'mime'=>$mime] = getimagesize($source);
    switch ($mime) {
        case 'image/gif':

            break;
        case 'image/jpeg':

            break;
        case 'image/png':

            break;
        default:

    }
```

Note that we've allowed for the three classic formats only. You can add others if you like.

There is all the difference between an image type and the file's extension. For example, an image saved or transmitted as the JPEG type may or may not have been saved with the traditional `.jpeg` or `.jpg` extension.

The operating system typically uses the file extension to determine the file's type. However, it may not be correct, especially when some malfeasant wants to inject something unpleasant. PHP can look beyond the extension and into the file itself.

For each image type, there is a corresponding function to load it called something like `imagecreatefrom...`, which returns a reference to the actual image data. You can store this in a variable called `$original`:

```
//  Load Image
    [0=>$sw, 1=>$sh, 'mime'=>$mime] = getimagesize($source);
    switch ($mime) {
        case 'image/gif':
            $original = imagecreatefromgif($source);
            break;
        case 'image/jpeg':
            $original = imagecreatefromjpeg($source);
            break;
        case 'image/png':
            $original = imagecreatefrompng($source);
            break;
        default:

    }
```

For a default case, you can use the `NULL` value: `default:`

```
        $original = null;
```

This can be used to indicate that the image wasn't loaded.

The function now looks like this:

```
function resizeImage($source, $destination, $size='160x120',
  $options=[]) {
    //  Set up
        ...
    //  Load Image
        [0=>$width, 1=>$height, 'mime'=>$mime]
          = getimagesize($source);
```

```
        switch ($mime) {
            case 'image/gif':
                $original = imagecreatefromgif($source);
                break;
            case 'image/jpeg':
                $original = imagecreatefromjpeg($source);
                break;
            case 'image/png':
                $original = imagecreatefrompng($source);
                break;
            default:
                $original = null;
        }
}
```

While we're here, we'll look at how to save the result next.

Saving the Resized Image

Just as with loading an image, PHP can save an image a number of different formats, including JPEG, PNG, and GIF formats, but, again, you'll need to choose the appropriate function.

To save an image as a file, we use a function called something like `imagepng(...)`, `imagejpeg(...)`, or `imagegif(...)`, depending on the file type. For example, `imagejpeg($image, $filename);` will save the image as a JPEG file.

If you omit the file name, PHP will output the raw image data to the screen. Of itself, this is useless, since your browser won't interpret it properly, being, as it were, mixed in with other data. However, in combination with additional code to produce the appropriate HTTP headers, this can be used to generate a live image for immediate display. This can be used, for example, to display live graphs or date-stamped images.

Similar to the `loadImage()` function, we'll use the `switch()` statement to choose between image saving functions. We already have the destination from the function parameter. We'll assume, for now, that the image to be saved is in a variable called `$copy`, which we'll work on next:

```
//  Save New Image
    switch ($mime) {
        case 'image/gif':
            imagegif($copy, $destination);
            break;
        case 'image/jpeg':
            imagejpeg($copy, $destination);
            break;
        case 'image/png':
            imagepng($copy, $destination);
            break;
        default:

    }
```

Here again, we use the `$mime` variable to choose between image saving functions. Later, we'll see this as an opportunity to change the saved image type.

The function now looks like this:

```
function resizeImage($source, $destination, $size='160x120',
$options=[]) {
    //  Set up
        ...
    //  Load Image
        ...

    ...

    //  Save New Image
        switch ($mime) {
            case 'image/gif':
                imagegif($copy, $destination);
                break;
            case 'image/jpeg':
                imagejpeg($copy, $destination);
                break;
            case 'image/png':
                imagepng($copy, $destination);
                break;
            default:

        }
}
```

The next step is to create the new image to be saved.

Creating the Blank Copy and Copying the Original

The important part of the job will be to create a new empty image of the right dimensions and to copy the original into it.

To create an empty image, we use `imagecreatetruecolor($width, $height)`. The function name reflects that we plan to allow for 24-bit color ("true color"), rather than a shallower color depth. This function requires the physical width and height of our new image. We can use the dimensions passed in the function call.

The resulting (blank) image will be stored in a variable called `$copy`:

```
function resizeImage($source, $destination, $size='160x120',
  $options=[]) {
    //  Set up
        ...
    //  Load Image
        ...

    //  Adjust Scale & Dimensions

    //  Create Blank Image
        $copy = imagecreatetruecolor($width, $height);
    ...

}
```

By default, the new image is actually filled in with black, which will be the background when we start adjusting the image dimensions. You can change that to another color if you like. We'll see how later on.

Copying the Original into the Image Copy

PHP has a number of functions to copy image data from one image to another. However, since you only want to create what is otherwise a simple copy of the image, the most suitable function is `imagecopyresampled()`, which will rescale and resample the copy nicely.

The `imagecopyresampled()` function has *ten parameters*, which is very flexible but somewhat overwhelming. They allow you to copy part of any image into any part of another image. This is great if you want to create a collage, but here we'll only need part of the flexibility. Nevertheless, all of the parameters must have a value.

To make the function call easier to read and maintain, we can take advantage of our ability to break a PHP statement over several lines.

Here are the ten parameters.

```
imagecopyresampled(
    destination_image, source_image,
    destination_x, destination_y, source_x, source_y,
    destination_width, destination_height, source_width,
    source_height
);
```

The function parameters specify the data of each image, the top-left corner ("origin") of each image, and the width and height of each image portion. The parameters are written in pairs for the destination and source, respectively.

When the time comes, we'll use the following code:

```
//  Copy the Original into the Smaller Image

    imagecopyresampled(
        $copy, $original,       // destination image,
                                   source image
```

```
        (int) $dx, (int) $dy, (int) $sx, (int) $sy,
        // left-top corner
        (int) $dw, (int) $dh, (int) $sw, (int) $sh
        // width, height
    );
```

The `(int)` expression before the variables *casts* the value to an integer. The `imagecopyresampled()` function complains if you give it decimals to work with, and we're likely to get decimals when we start scaling the copy later.

This function breaks a convention for copying data. Normally, you would copy *from* somewhere *to* somewhere else. In this function, you copy *to* the copy *from* the original. The parameters are

- The (empty) copy from the source image
- The coordinates (x,y) of the top-left corners of the copy and source images
- The dimensions (width, height) of the copy and source images

The image variables (`$copy`, `$original`) and the source image dimensions (`$sw`, `$sh`) have already been defined.

The rest of the variables will be defined in the following steps:

- We will always copy the whole of the source image. This means from the top-left corner (`0`, `0`) and using its full width and height (`$sw`, `$sh`), as read when loaded. For the source origin, we can use

  ```
  $sx = $sy = 0;
  //  $sw and $sh already defined
  ```

- The destination variables will need adjusting later, but for now we'll specify the whole of blank image copy. This means using (0, 0) for the origin and ($width, $height) for the size.

We'll copy them into new variables to make the code a little more intuitive and to make adjusting them easier:

```
$tx = $ty = 0;
[$tw, $th] = [$width, $height];
```

The code now looks like this:

```
function resizeImage($source, $destination, $size='160x120',
  $options=[]) {
    ...

    //  Create Blank Image
        ...

    //  Adjust Scale & Dimenstions
        $sx = $sy = 0;
        //  $sw and $sh already defined
        $dx = $dy = 0;
        [$dw, $dh] = [$width, $height];

    //  Copy Original into New Image
        imagecopyresampled(
          $copy, $original,          // destination, source
          (int) $dx, (int) $dy, (int) $sx, (int) $sy,
          // left-top
          (int) $dw, (int) $dh, (int) $sw, (int) $sh
          // width, height
        );
```

```
    //  Save New Image
    ...
}
```

At this point, you should have a working function, and you can call the function as follows:

```
resizeImage($source, $destination, '160x120');
```

The function isn't perfect yet, as it will distort the image copies, but at least you can get it ready for testing.

Note that although there is a default copy size, it is always safe to include them anyway, making it easier to change your mind.

Padding Adjustments for Distorted Copies

If your copy shape is not the same as the original image, then you will see a distorted image as it is stretched and squeezed into the new shape, as in Figure D-1.

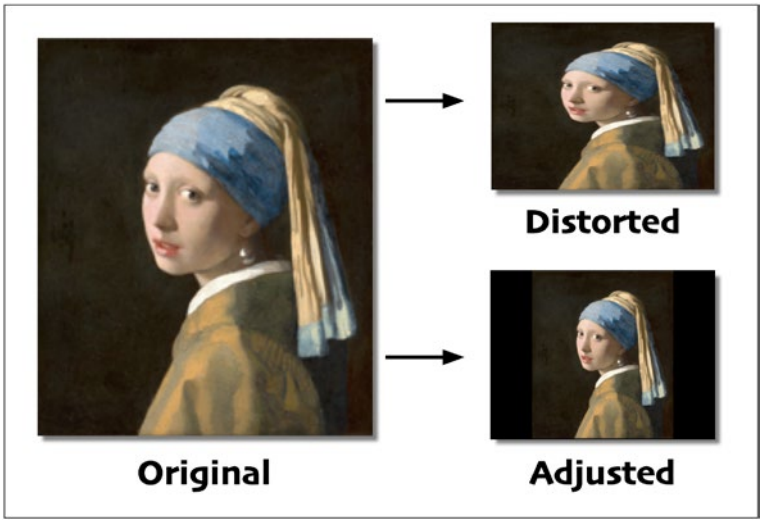

Figure D-1. *Distorted Copy*

Clearly, it would be better for the image to keep its shape inside the copy like the adjusted version.

To allow for a differently shaped image, you will need to adjust the part of the copy where the original will be copied into, so that the original shape is maintained. The difference in shape will be padded with the default black background or whatever color you decide on later.

You can see the idea in Figure D-2.

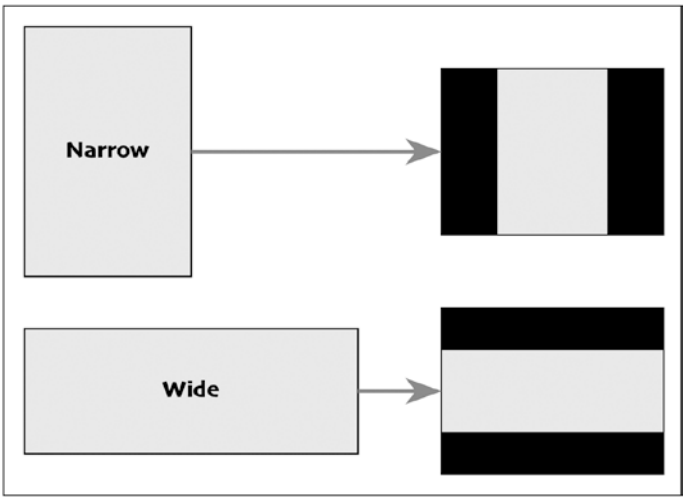

Figure D-2. *Padding Copies*

To create this padding, the destination corner point will need to be adjusted, as well as either the destination width or the destination height.

The calculation will be as follows:

- If original is thinner than copy
  - Adjust copy width
  - Adjust copy left coordinate (x)

- Else if original is wider than copy
  - Adjust copy height
  - Adjust copy top coordinate (y)
- Else (same shape)
  - Do nothing

Note that we only need to adjust *one* of the copy coordinates and *one* of the copy dimensions.

Image Shapes

At some point in your schooling, you would have learned about ratios. In case anybody is asking "why do we need to learn about ratios," you can say they help in rescaling images in PHP.

The term **aspect ratio** refers to the ratio of the width to the height of an image. You can calculate it as

$$aspect = \frac{width}{height}$$

If one image is thinner than another, then its aspect ratio will be less. If it is wider, then its aspect ratio will be greater. If the images are the same shape (even at different sizes), then the aspect ratio will be the same.

In the adjustment part of the `imageResize()` function, you can include the following:

```
//  Adjust Scale & Dimenstions
    ...
    if($sw/$sh < $dw/$dh) {     //  original is narrow

    }
    elseif($sw/$sh > $dw/$dh) { //  original is wide

    }
    //  else do nothing
```

If the two shapes are the same, we will do nothing.

Adjusting the Size and Origin

The adjusted copy dimensions need to have the same ratio as the source dimensions:

$$\frac{dw}{dh} = \frac{sw}{sh}$$

The adjusted origin will need to start in the middle and move back *half* of the copy width.

For now, we'll focus on a narrower original image. This means adjusting the width and x-coordinate of the copy. The y-coordinate will stay at `0`, and the height will be the height of the copy. Figure D-3 shows how this is going to work.

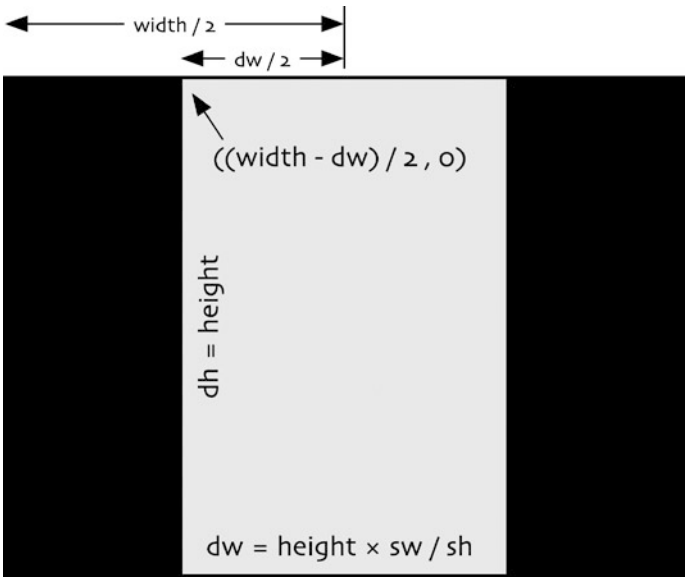

Figure D-3. *Adjusting the Width and Origin*

For the narrow version, we need to calculate the new width. This gives us

$$dw = dh \times \frac{sw}{sh}$$

In PHP, this becomes

```
$dw = $dh * $sw / $sh;
```

As for the origin, conceptually we move to the horizontal middle of the copy and move back half of the copy width:

```
$dx = $width/2 - $dw/2;
```

In practice, it's better to refactor this:

```
$dx = ($width - $dw) / 2;
```

Adding this to our function code, we get

```
//  Adjust Scale & Dimenstions
    ...
    if($sw/$sh < $dw/$dh) {     //  original is narrow
        $dw = $dh * $sw / $sh;
        $dx = ($width - $dw) / 2;
    }
    elseif($sw/$sh > $dw/$dh) { //  original is wide

    }
    //  else do nothing
```

For a wider original image, it is the height which needs to be adjusted, as well as the y-coordinate of the origin. Following the same reasoning as before, we get

```
$dh = $sh * $width / $sw;
$dy = ($height - $dh) / 2;
```

Again, adding to our function code

```
//  Adjust Scale & Dimenstions
    ...
    if($sw/$sh < $dw/$dh) {     //  original is narrow
        $dw = $dh * $sw / $sh;
        $dx = ($width - $dw) / 2;
    }
    elseif($sw/$sh > $dw/$dh) { //  original is wide
        $dh = $sh * $width / $sw;
        $dy = ($height - $dh) / 2;
    }
    //  else do nothing
```

If you try this now with various images, you should see padding added appropriately to maintain the shape of the copy.

The Complete resizeImage Function (So Far…)

You can view the completed code so far in a file called `resizeImage-sofar.php`. Here is an abbreviated version of the code, so you can compare what you've got:

```
function resizeImage($source, $destination=null,
$size='160x120',
    $options=[]) {
        //  Set up
        $pattern = '/^\s*(\d+)\s*[xX]\s*(\d+)\s*$/';
        if(preg_match($pattern, $size, $matches)) {
            [, $width, $height] = $matches;
        }
        else {
            [$width, $height] = [160, 120];
        }

    //  Load Image
        [0=>$sw, 1=>$sh, 'mime'=>$mime] = getimagesize
        ($source);
        switch ($mime) {
            ...
        }

    //  Adjust Scale & Dimensions
        $sx = $sy = 0;
        //  $sw and $sh already defined
        $dx = $dy = 0;
        [$dw, $dh] = [$width, $height];
```

```
        if($sw/$sh < $dw/$dh) {     //  narrow
            $dw = $height * $sw / $sh;
            $dx = ($width - $dw) / 2;
        }
        else {                      //  wide
            $dh = $sh * $width / $sw;
            $dy = ($height - $dh) / 2;
        }

    //  Create Blank Image
        $copy = imagecreatetruecolor($width, $height);

    //  Copy Original into New Image
        imagecopyresampled(
            $copy, $original,
            (int) $dx, (int) $dy, (int) $sx, (int) $sy,
            (int) $dw, (int) $dh, (int) $sw, (int) $sh
        );

        switch ($mime) {
            ...
        }
    }
```

The next job will be to handle variations.

Options

We've made a few assumptions about how the result should look, but we can add a few options to make the function more flexible. Here are some possibilities:

- Allow the destination name to be omitted.
- Change the background color.

- Change the copy image type.
- Allow a transparent background.
- Save a copy without padding.

It's easy to get carried away with flexibility so that the job is never finished, but the preceding options shouldn't be too hard to implement. Apart from the first possibility earlier, we'll use an `$options` array variable for the other possibilities, so as not to build an overwhelming collection of parameter variables, like the `imagecopyresampled()` function.

Omitting the Destination Name

You can change the function parameters to make the `$destination` variable optional, defaulting to `null`:

```
function resizeImage($source, $destination=null,
$size='160x120',
    $options=[]) {

}
```

The only problem with traditional optional parameters is that they must be specified in order. So, for example, if you want to specify your own `$size` value, which is very likely, you can't skip the `$destination` value, and you'll probably set it to `null` anyway.

In the section on named parameters, you'll see how this can be improved upon.

In any case, if the `$destination` value is `null`, then we can build our own destination name from the original name and the `$size` value.

We can add the following code to the section which saves the image:

```
//  Save New Image
    //  Missing Destination
        if(!$destinaton) {
            //  construct new name
        }
```

The new name can take the following form:

```
{directory}/{filename}-{width}x{height}.{extension}
```

where the directory, file name, and extension come from the original file name and the dimensions come from the `$width` and `$height` variables.

To get the components of the original path and name, you can use the `pathinfo()` function. This gives us an array with four components. You can then use array dereferencing to extract the components:

```
//  Save New Image
    if(!$destinaton) {
        ['dirname'=>$dirname, 'basename'=>$basename,
            'extension'=>$extension, 'filename'=>$filename]
        = pathinfo($source);

    }
```

We won't really need the `$basename` variable, since it's just a combination of the `$filename` and `$extension` values, so we can leave it out:

```
//  Save New Image
    if(!$destinaton) {
        ['dirname'=>$dirname, 'extension'=>$extension,
            'filename'=>$filename]
          = pathinfo($source);
    }
```

We can now reconstruct the name to include the dimensions:

```
//  Save New Image
    if(!$destination) {
        ['dirname'=>$dirname, 'extension'=>$extension,
            'filename'=>$filename]
          = pathinfo($source);
        $destination
          = "$dirname/$filename-{$width}x{$height}.$extension";
    }
```

Note that we've wrapped the `$width` and `$height` variables in braces. In the case of the `$width` variable, this is to stop it from running on to the `x` which would have been mistaken for part of the variable name. You can do that to all of the variables, but it's only included for `$height` for symmetry.

Using the Options Array

The `$options` array might look like this:

```
$options = [
    'background' => [165,62,22],    //  r,g,b
    'type' => 'png',
    'method' => 'trim',
];
```

The point of the `$options` array is that not all options may be present. You can take a simplistic approach and use two steps for each option:

- Test whether the option is present.
- If so, use the option.

In PHP, this would look like this:

```
if(isset($options['...'])) {
    //  use option
}
```

Another approach is to have another array of defaults and merge the two arrays. We can then go ahead and use the resulting array:

```
$defaults = [ ... ];
$options = [...$defaults, ...$options];
```

The spread syntax (`...$variable`) destructures the array variables into keys and values, and the `[ ... ]` expression then constructs a new array from the results. If the newer keys are the same as the previous keys, they will replace them. Prior to PHP 7.4, you would use the `array_merge()` function:

```
$defaults = [ ... ];
$options = array_merge($defaults, $options);
```

At the beginning of the `resizeImage()` function, we can now process the options array:

```
function resizeImage($source, $destination=null,
$size='160x120',
    $options=[]) {

    //  Setup
        ...

        $defaults = [
            'background' => [0,0,0],    //  r,g,b (black)
            'type' => null,             //  use same as source
            'method' => 'pad',          //  pad copy
        ];
```

```
        $options = [...$defaults, ...$options];
        //  $options = array_merge($defaults, $options);
}
```

We can now accommodate the rest of the options.

Setting a Background Color

The default background color when you create a new image is black, which is a reasonable way to start. You can set the background color to any RGB value, but it's a little complicated.

The `imagecolorallocate($image, $red, $green, $blue)` function creates an identifier for an image based on the red, green, and blue values you give it. You then use the result to fill in the created image.

The `imagefill($image, $x, $y, $colour)` function fills in the image with the color from the corner specified.

For the background color, we'll use an array of three values, each from 0 to 255. They will be the red, green, and blue values, respectively. For example:

```
resizeImage($source, $destination, $size, [
    'background' => [164, 62, 22]
]);
```

In our function, we can add the following:

```
//  Create Blank Image
    $copy = imagecreatetruecolor($width, $height);

//  Background Colour
    $background
      = imagecolorallocate($copy, ...$options['background']);
    imagefill($copy, 0, 0, $background);
```

Here again, we use the spread operator (`...`) which destructures the array into three separate values, which is what the `imagecolorallocate()` function requires.

You can now specify a background color or leave it at the default black.

Changing the Saved Image Type

As it stands, the resized copy will be the same type as the original, which is a reasonable assumption. It's not hard, however, to change that with one line of code.

The real problem will be with the file name. Technically, it's not necessary, but the file extension should match the MIME type—otherwise, some software will get terribly confused.

In this case, we'll do the following:

- Begin with the file extension of the `$destination` variable.
- If `$options['type']` has been set, replace the extension with this value.
- Use the resulting extension to determine the MIME type.

First, we'll destructure the destination file name, similarly to what we did with the source name:

```
//  Save New Image
    //  Missing Destination
        ...
    //  Check Mime Type
        ['dirname'=>$dirname, 'basename'=>$basename,
            'extension'=>$extension, 'filename'=>$filename]
          = pathinfo($destination);
```

We can then create an array of our accepted file extensions and their corresponding MIME types:

```
//  Check MIME Type
    ['dirname'=>$dirname, 'basename'=>$basename,
        'extension'=>$extension, 'filename'=>$filename]
      = pathinfo($destination);
    $imageTypes = ['gif'=>'image/gif','jpeg'=>'image/jpeg',
        'jpg'=>'image/jpeg','png'=>'image/png'];
```

You'll see that the array has two file extensions for JPEG files, both of which are common; they still have the same MIME type.

Next, use the `$options` array to override the extension:

```
//  Check MIME Type
    ...
    if($options['type']) $extension = $options['type'];
```

We can now reconstruct the destination file name with the (possibly) adjusted extension:

```
//  Check MIME Type
    ...
    if($options['type']) $extension = $options['type'];
    $destination = "$dirname/$filename.$extension";
```

Finally, we'll use the extension to set the new MIME type:

```
//  Check MIME Type
    ...
    if($options['type']) $extension = $options['type'];
    $destination = "$dirname/$filename.$extension";
    $mime = $imageTypes[$extension];
```

Remember, PNG images take much more space than JPEG images, so you're more likely to change from PNG than the other way round. However, if you want to include transparency, then PNG is definitely the way to go.

Setting a Clear Padding Background

PNG images optionally include an alpha layer, which is a mask describing what parts of the image are solid, transparent, or anything in between. We'll use that to set a clear padding background.

We'll use the `$options['method']` value to decide whether we want to do this. The three values will be

- `pad`: This is the default padding with a background color.
- `clear`: This pads the copy with a clear background; we'll implement this now.
- `trim`: This will trim the copy to the actual image without padding; we'll do that in the next section.

To implement a clear background, we'll need a variation on the `imagecolorallocate()` function we used earlier to generate the fill color. First, we'll use a `switch()` statement to decide whether to do this:

```
//  Background Colour
    switch($options['method']) {
        case 'pad':
            $background
              = imagecolorallocate($copy,
                ...$options['background']);
            imagefill($copy, 0, 0, $background);
            break;
        case 'clear':

            break;
    }
```

We use a `switch()` in case there are more methods at some point in the future. Remember the default `pad` has already been set.

To implement transparency, we'll need a different function, `imagecolorallocatealpha()`, to set the color with alpha and to set a flag which causes the image to be saved with the alpha mask later:

```
//  Background Colour
    switch($options['method']) {
        case 'pad':
            ...
        case 'clear':
            $background
              = imagecolorallocatealpha($copy, 0, 0, 0, 127);
            imagesavealpha($copy, true);
            imagefill($copy, 0, 0, $background);
            break;
    }
```

There's one more step, though. The only classic image format which supports alpha masks is the PNG format, so you'll need to override the extension, which sets the MIME type:

```
//  Check MIME Type
    ...
    if($options['type']) $extension = $options['type'];
    if($options['method']=='clear') $extension = 'png';

    $destination = "$dirname/$filename.$extension";
    $mime = $imageTypes[$extension];
```

If you're building a neatly spaced gallery, it helps to have all of your copies the same shape and size. Having a clear padding background, however, allows your gallery to have whatever background you like.

Trimming the Image Copy

The code has assumed that you want your copy to be a fixed shape and size, which is why it's padded to fit the shape. You may want to dispense with the padding, so that the copy is the same shape as the original.

For this, we'll use the option

```
$option['method'] = 'trim';
```

With this option, you'll need to make two changes:

- The copy dimensions of the copy image should be those of the adjusted destination. We'll still accept both width and height and let the dimensions be the best fit.

  However, there may be a problem with the dimension values, which we'll need to allow for.

- The origin for the destination should be set to (0, 0), which is the top-left corner.

For the copy image, we can make the following change:

```
//  Create Blank Image
    switch($options['method']) {
        case 'trim':
            $copy = imagecreatetruecolor((int) $dw, (int) $dh);
            break;
        case 'pad':
        case 'clear':
        default:
            $copy = imagecreatetruecolor($width, $height);
    }
```

Note that we've cast the `$dw` and `$dh` variables to integers. Again, the `imagecreatetruecolor()` function doesn't like decimals.

In the `switch()` statement, we've allowed the `pad` and `clear` options to fall through the default which is to create an empty image using the requested size.

For the destination origin coordinates, we had already set them to 0, but changed them when adjusting for the shape. We'll add a condition to only do that if the option isn't set to `trim`:

```
//  Adjust Scale & Dimensions
    ...
    if($sw/$sh < $dw/$dh) {     //  narrow
        $dw = $height * $sw / $sh;
        if($options['method'] != 'trim') $dx =
        ($width - $dw) / 2;
    }
    else {                      //  wide
        $dh = $sh * $width / $sw;
        if($options['method'] != 'trim') $dy =
        ($height - $dh) / 2; }
```

You might also decide that you'd like to use a percentage instead of an enclosing rectangle if you're going to trim the result. That's not hard, but we'll leave that for some future enhancement.

Using Named Parameters (PHP 8)

You've seen that functions like `imagecopyresampled()` not only have an unwieldy name, but an absurd number of required parameters which must be supplied in a counterintuitive order. You'll see that in many PHP functions, but that's no reason to follow their example.

Our `resizeImage()` function has fewer parameters and allows options to be added in an additional array. Starting in PHP 8, there's an alternative approach.

Named parameters allow you to specify the name of the parameter variable, rather than relying on its position. For example, you can write the following alternative to the code we wrote before:

```
imagecopyresampled(
    dst_image: $copy, src_image: $original,
    dst_x: (int) $dx, dst_y: (int) $dy,
    src_x: (int) $sx, src_y: (int) $sy,
    dst_width: (int) $dw, dst_height: (int) $dh,
    src_width: (int) $sw, src_height: (int) $sh
);
```

Of course, that's not much of an improvement. However, if you want to follow the more conventional pattern of putting the source first, you can:

```
imagecopyresampled(
    src_image: $original, dst_image: $copy,
    src_x: (int) $sx, src_y: (int) $sy,
    dst_x: (int) $dx, dst_y: (int) $dy,
    src_width: (int) $sw, src_height: (int) $sh,
    dst_width: (int) $dw, dst_height: (int) $dh
);
```

Given the awkward parameter names and the fact that they're all required, it's still not so much of an improvement. However, when it comes to optional parameters, that's another story.

For example:

```
function test($a, $b=3, $c=4) {
    print $a + $b*$c;
}

test(5, c: 3);
```

Here, we have optional variables $b and $c. Using named parameters gives us the opportunity of setting $c without bothering with $b. Note that with the first parameter, $a, it's not optional. You can either set it by name in any order or by position in the first position only.

Named Parameters in the resizeImage() Function

To use named parameters in the resizeImage() function, we can make the following changes.

First, change the function definition to use separate optional parameter variables instead of the $options array:

```
function resizeImage(
    $source, $destination=null, $size='160x120',
    $background=[0,0,0], $type=null, $method='pad'
) {
    ...
}
```

Having set them here, we no longer need to bother merging arrays. You can delete or comment out the appropriate code:

```
function resizeImage(
    $source, $destination=null, $size='160x120',
    $background=[0,0,0], $type=null, $method='pad'
) {
    ...
    //  Set up
        ...

/*
```

```
    $defaults = [
        'background' => [0,0,0],    //  r,g,b (black)
        'type' => null,             //  use same as source
        'method' => 'pad',          //  pad copy
    ];
    $options = array_merge($defaults, $options);
*/

    ...
}
```

The next part is to replace all the references to the `$options` array with the simple variables. First, with the scale and dimensions:

```
//  Adjust Scale & Dimensions
    ...

    if($sw/$sh < $dw/$dh) {     //  narrow
        $dw = $height * $sw / $sh;
        // if($options['method'] != 'trim') $dx =
        ($width - $dw) / 2;
        if($method) $dx = ($width - $dw) / 2;
    }
    else {                      //  wide
        $dh = $sh * $width / $sw;
        // if($options['method'] != 'trim') $dy =
        ($height - $dh) / 2;
        if($method != 'trim') $dy = ($height - $dh) / 2;
    }
```

Next with the blank image and background color:

```
//  Create Blank Image
    switch($method) {
        ...
    }

//  Background Colour
    switch($method) {
        case 'pad':
            $background
              = imagecolorallocate($copy, ...$background);
            imagefill($copy, 0, 0, $background);
            break;
        case 'clear':

            break;
    }
```

And finally with the MIME type:

```
//  Check MIME Type
    ['dirname'=>$dirname, 'basename'=>$basename,
        'extension'=>$extension, 'filename'=>$filename]
      = pathinfo($destination);
    $imageTypes = ['gif'=>'image/gif','jpeg'=>'image/jpeg',
        'jpg'=>'image/jpeg','png'=>'image/png'];

    if($type) $extension = $type;
    if($method == 'clear') $extension = 'png';
```

There may be other reasons why you might prefer to keep the options in an array as before. One reason would be if you are working in an older version of PHP and don't have the option of updating it.

splitSize

The `resizeImage()` includes some code to split a string like `180 x 240` into two values. We'll need to do that on other occasions as well, such as deciding on the `width` and `height` attributes of an `img` element, so we have a stand-alone function to do that.

You have used the `splitSize()` function a few times. In its simplest form, you can write the function as

```
function splitSize(string $size) {
    $pattern = '/^\s*(\d+)\s*[xX]\s*(\d+)\s*$/';
    if(preg_match($pattern, $size, $matches))
        [, $width, $height] = $matches;
    else [$width, $height] = [null, null];

    return [$width, $height];
}
```

This is basically the same code as that in the `resizeImage()` function, except that the `else` part gives us a pair of `null`s, rather than default dimensions.

In the actual function, there's a little more. For your convenience, you have the option of embedding those dimensions in a string like

```
width="180" height="240"
```

This makes it easier to include in the `img` tag later on.

The complete function looks like this:

```
function splitSize(string $size, $img=false) {
    $pattern = '/^\s*(\d+)\s*[xX]\s*(\d+)\s*$/';
    if(preg_match($pattern, $size, $matches))
        [, $width, $height] = $matches;
    else [$width, $height] = [null, null];
```

```
    if($img) {
        if($width && $height)
            return sprintf('width="%s" height="%s"', $width,
            $height);
        else return '';
    }
    else return [$width, $height];
}
```

In this modified version

- There is an additional optional parameter `$img` which defaults to `false`. If it's not used, it will return the array with two values as before. If used, it will return the combined string instead.
- If `$img` is `true`, we also need to test whether the two dimensions have been set. If so, we can return the string using the `sprintf()` function. If not, we simply return an empty string.

We could have used this function in the `resizeImage()` function, but the code was duplicated simply to make the `resizeImage()` function more self-contained.

unzip Function

PHP has a `ZipArchive` class which allows us to work with ZIP files. In principle, it should be simple to extract files, but it's complicated by the fact that the files we want may or may not be in a folder. For that reason, we have a function to do that less simply.

We briefly described the process in Chapter 4, but here is more detail.

The function is called `unzip()` and broadly looks like this:

```
function unzip($source, $destination) {

}
```

The `$source` parameter is a reference to the ZIP file, and the `$destination` is the directory where the files will be going.

That should do the job, but in the process we'll be getting a list of file names. Two of them actually, and that might be useful. Or it might not. However, it doesn't hurt to make them available, which we'll do in a `return` statement.

To begin with, we create a `ZipArchive` object, open the file, and, later, close it:

```
function unzip($source, $destination) {
    $zip = new ZipArchive;
    $zip -> open($source);

    //  extract files

    $zip -> close();
}
```

However, instead of using the built-in `extractTo()` method, we need to iterate through the contents of the ZIP file to get at the individual files. To do this, we need two things:

- The `numFiles` property has the number of files or directories in the archive. Note that a directory is treated as a file.
- The `getNameIndex()` method extracts the *name* of one of the files, by index number.

To iterate through the archive, we use a `for()` statement:

```
$zip = new ZipArchive;
$zip -> open($source);

for($i = 0; $i < $zip -> numFiles; $i++) {
    //  extract the file
}

$zip -> close();
```

Within each iteration, get the name of the file. The name will be the full path name including the enclosing directory, if any, so we'll need to deal with that in a moment. For now, we'll test whether it's the name of a directory by testing its last character:

```
for($i = 0; $i < $zip -> numFiles; $i++) {
    $file = $zip -> getNameIndex($i);
    if($file[-1] == '/') continue;
}
```

- We use the counter number to reference each file name, using `getNameIndex()`.
- To get a single character from a string, we can use a notation which treats the string as an array of characters. The negative index counts from the end. Using `$file[-1]` means the first character from the end.
- If the last character is the forward slash (/), it's a directory name. Note that this is true even on Windows which prefers the backslash (\).
- If it's a directory, skip the rest and move on to the next one. The continue statement basically moves to the end of the block.

We'll now want to copy the file out of the archive into the real destination. For our particular case, we want to dispense with the internal path and just use the file name. We do that with the `basename()` function:

```
for($i = 0; $i < $zip -> numFiles; $i++) {
    $file = $zip -> getNameIndex($i);
    if($file[-1] == '/') continue;
    $name = basename($file);
}
```

Having got the base name, we'll do the copy:

```
for($i = 0; $i < $zip -> numFiles; $i++) {
    $file = $zip -> getNameIndex($i);
    if($file[-1] == '/') continue;
    $name = basename($file);
    copy("zip:// … #$file", " … /$name");
}
```

The `copy(from, to)` function, as the name suggests, copies the file. The special odd-looking format `zip:// ... #$file` copies from what was a virtual file in the ZIP archive. Here, we use the `$file` and `$name` values from the previous lines. The `from` part of the function references the original ZIP file; the `to` part of the function references where it's going.

At this point, there are two versions of the file name—the original and the simplified one. That might be useful, so we'll collect them in two arrays:

```
function unzip($source, $destination) {
    ...

    $files = [];
    $names = [];
```

```
    for($i = 0; $i < $zip -> numFiles; $i++) {
        $file = $zip -> getNameIndex($i);
        if($file[-1] == '/') continue;
        $name = basename($file);
        copy("zip:// ... #$file", " ... /$name");

        $files[] = $file;
        $names[] = $name;
    }

    $zip -> close();
}
```

Here, we've initialized two arrays before the `for()` block and added the `$file` and `$name` values inside the `for()` block.

Functions don't need to have a return value. In PHP, a function returns `null` anyway, unless you return something else. That might be a wasted opportunity, so we'll return the two arrays, just in case anyone's interested:

```
function unzip($source, $destination) {
    ...

    $files = [];
    $names = [];

    for($i = 0; $i < $zip -> numFiles; $i++) {
        ...
    }

    $zip -> close();

    return ['files' => $files, 'names' => $names];
}
```

The return value is an associative array to make it more understandable.

Markdown to HTML

The markdown language is easy enough to write in, but, in the end, it's not a web language. The next step is to convert markdown to HTML and let the CSS manage the presentation.

There are many markdown interpreters, both in JavaScript and in PHP, as well as many others in other languages. The `md2html()` function is a very simple interpreter written in PHP.

The code was inspired by **John de Plume**, a.k.a. **jbroadway**. You can see the original on GitHub: https://github.com/jbroadway/slimdown.

Unlike many other interpreters, this interpreter simply uses regular expressions to replace patterns, rather than a true language interpreter.

Regular expressions are special patterns used to analyze strings. Using regular expressions, you can define a pattern to be matched and how you might replace parts of the string. There's no official regular expression language, and there are variations, but most of them start off the same way.

Regular expressions are very concise and can be very powerful. They're also *very* hard to read, and we won't be going into detail here.

The most comprehensive version of regular expressions comes from the Perl programming language, and other interpreters often aim to be Perl-compatible. JavaScript has regular expressions, while PHP supports them via a library.

Regular expressions have limitations in what they can do, and this particular interpreter is not as rich or as powerful as some others. However, it's certainly good enough for our blog articles.

This is my take on the interpreter. We won't go into the full details, but enough to get a feeling for how it works and how you might modify it.

The function makes use of two PHP regular expression functions:

- `preg_replace(pattern, replacement, original)` replaces parts of the original string which match the pattern with a replacement.
- `preg_replace_callback(pattern, function, original)` does something similar, but uses a function to work out what the replacement should be.

Here are two examples, taken from the function.

Anchors

The markdown code for an anchor is `[text](reference)`, which translates to `<a href="reference">text</a>`.

The code to do the translation is

```
$text = preg_replace(
    '/\[(.+)\]\((.+)\)/',
    '<a href="$2" target="_blank">$1</a>',
    $text
);
```

The `preg_replace()` function replaces part of the text which matches a pattern.

This regular expression is complicated by the fact that the square brackets and parentheses are *both* special characters in regular expressions. To treat them as ordinary characters, you need to escape them—that is, precede them with a backslash (\).

For the first part, we're looking for something inside square brackets (`\[ ... \]`). The backslashes treat the square brackets as literal characters. Inside, we look for at least one character (`.+?`—the dot is any character, the plus means at least one, and the question mark means don't go too far). The result is captured as a group (`( ... )`).

The same sort of pattern matches the second part, except that we're looking inside parentheses (`\( ... \)`). Again, the literal parentheses need to be escaped.

Part of the regular expression includes what are known as **capture groups**: references to part of the matched pattern. Those references are `$1` and `$2`, in the order they're found. The reason they're reversed is that markdown puts the text before the reference, while HTML puts the reference before the text.

The `target="_blank"` is a nonstandard addition for the sake of the blog article. It causes the link to open in a new tab or window.

Headings

HTML headings are `h1` to `h6` to indicate the level of the heading. There are two ways to mark a heading in markdown, but here we're using the hash method: the line begins with one or more hash characters (#) and some space. The number of hashes is the heading level.

Translating headings is a little more complicated than anchors because of the varying heading level. For that reason, we use `preg_replace_callback()`, which allows you to include a function rather than a simple replacement; the `return` value of the function is the actual replacement value.

The part of the code is

```
$text = preg_replace_callback(
    '/^(#{3,})(.*)/m',
    function($data) {
```

```
        [, $level, $text] = $data;
        return sprintf('<h%1$s>%2$s</h%1$s>',
            strlen($level), trim($text));
    },
    $text
);
```

The regular expression looks for a string at the beginning of the line (^) and a number of hashes. In this case, it's looking for three or more hashes #{3,}. Markdown will happily work with one or more hashes, but, for the blog, we reserve the first two levels for the web page. After that, we want the rest of the text (.*).

Putting the match expressions inside parentheses ((#{3,}) and (.*)) causes them to be captured for further use.

The second parameter is an **anonymous function**, also called a **closure**. It's a function which hasn't been defined separately and, in this case, isn't even given a name. It doesn't need to be that way. You can also use a predefined or built-in function if there's one which you find suitable.

Miscellaneous Functions

There are also a few minor functions, which basically wrap other functions into something more convenient.

The MimeType() Function

Sometimes, you need to know the type of file you're working with. Sometimes, that's for uploaded files, but it might also be for files which already exist around the place.

For uploaded files, the $_FILES array does include the type value, but that's not 100% reliable. For other files, you don't even have that.

PHP has a class called `finfo`. You can learn some things about the file that way, but here the interesting part is the MIME type.

Unfortunately, there's no objective way to work out what type of file you've got. The normal method is to look inside the file itself and look for certain telltale patterns. The function relies on a file called `magic.mime` for this. Fortunately, the `finfo` object will do that for you.

Using `finfo` isn't too difficult, but it's convenient to wrap it inside a simple function:

```
function MimeType($filename) {
    $finfo = new finfo(FILEINFO_MIME_TYPE);
    return $finfo ->    file($filename);
}
```

The `$finfo` object is created to specifically extract MIME type information, and the `file()` method does the extracting.

This is a very roundabout and counterintuitive way of getting this sort of information, which is why it's convenient to wrap it inside a more natural function.

The printr() Function

PHP has a built-in function called `print_r()` which can be used to output complex structures, such as an array. You don't have much control over its appearance because it's really only meant to be used while you're developing or troubleshooting.

To make the structure easier to understand, the output includes line breaks and indentation.

The only problem is that HTML ignores all line breaks and spacing, *unless* the content is inside a `pre` element or another element specifically designed to support the text layout.

For convenience, the library includes a function called `printr()` which simply outputs the `pre` tags around the actual `print_r()` output, so that you can see the layout within the rest of the HTML:

```
function printr($data) {
    print '<pre>';
    print_r($data);
    print '</pre>';
}
```

Index

A

B

M. Simon, *An Introduction to PHP*, https://doi.org/10.1007/979-8-8688-0177-8

C

D

E

F

G

H

I

J, K

L

M

N

O

P, Q

R

S

T

U

V

W, X, Y

Z