|  |  |
| --- | --- |
| **w1fma PHP Coursework Report** | **jbliss02** |
| [**http://titan.dcs.bbk.ac.uk/~jbliss02/w1fma/index.php**](http://titan.dcs.bbk.ac.uk/~jbliss02/w1fma/index.php) | |

**Classes**

The first structural decision I had to take was how to organise the code that managed and manipulated the images. Although the actions on a file being uploaded were distinctly different to that of an existing file being displayed there were sufficient common elements that would have led to code duplication if they had been implemented independently. Therefore I created 4 classes:

|  |  |
| --- | --- |
| ***pic*** | A base class that contained common properties and methods to all images |
| ***uploadedPic*** | Inherits from ***pic***. Includes properties and methods that pertain to the management of an image being uploaded |
| ***downloadedPic*** | Inherits from ***pic***. Includes properties and methods that pertain to the management of an existing image on the server |
| ***picCollection*** | A class that holds a collection of ***downloadedPic*** classes and appropriate methods to instantiate and manage that collection. |

**File upload**

When a user uploads a file we first check that the upload was successful (as in it has been successfully loaded into the temporary directory). Once this has been complete the file type is checked to ensure it is of type IMAGETYPE\_JPEG. This type check is in the uploaded Pic class. It may be sensible to move this into config.php and make it an array so it is simple to add additional supported file types.

Once the file and file type have been validated a name is created for the new file, the existing name is not used. The name is a concatenation of two fields:

1. A current timestamp, denoted by the PHP variable time()

2. An incremental number in the database table imgId which is assigned on a per image basis

I felt that with the combination of these two fields the chances of a file getting the same name where next to nothing, and certainly within an acceptable tolerance.

I then had to determine whether the image needed to be resized to fit the size constraints. One challenge I encountered was that a single resize was not always sufficient; I had to loop the resizing algorithm until both the height and width where allowable.

The penultimate step was saving the image. If the image had to be resized then a new image was created on the fly so this was saved using the imagejpeg function. If the temporary image did not need to be resized then we could copy the temporary file. Both actions throw an error if the copy / save is not successful.

The final step was saving this information to the database. The database classes contained sufficient error handling to deal with any unexpected errors. It was important to add the database call last so that we were sure that if the data was in the database that the file was sound and had been saved correctly.

**Viewing files**

I decided against creating a thumbnail image each time a file was saved and instead created the thumbnail on the fly. This saved creating another database entry for the thumbnail and allows the size of the thumbnail to be changed in future without having to regenerate the thumbnail.

The disadvantage of this approach was computing resource to generate the thumbnail on each visit to index.php. This would only be noticeable with a number of thumbnails to create at any one time and this could be solved by introducing a paging element to the application.

When index.php is loaded the application gets a list of all files from the webservice, it then creates an object for each adding them into a collection object. At this point the system checks whether the file and folder still exists, if it does not then the object is not created for that particular file.

**The web service**

There has been some duplication of code in the db.php file that can be found in /webservice and /classes. This duplication of code has been allowed so that the web service can work in isolation to the main program and can be moved, without affecting the main program.

Once the web service (effectively the code in /webservice) was moved then the end point would just need to be changed in the config.php file.

The duplicated class contains the database connection information; the application accesses the database directly when an image is uploaded, the web service accesses the database to find out information about the database. In a production environment I would consider uploading images through the web service as well so the application has no reason to connect to the database.

**Portability**

The programmatic creation of html has been limited to the methods within the ***functions.php***

The web service has no dependencies on the main application and as such can be moved. The application only references the webservice url through the config.php file.

All text outputs are contained within lang/en.php to allow simple changes or extension of the application to another language.

The database class inherits from the mysqli class so the database engine can easily be changed.

The location that images are uploaded to is contained within the config.php file