Jesuitas
Logroño

SERVER-SIDE WEB PROGRAMMING UNIT3: STORING INFORMATION WITH DATABASES

CFGS - DAW 2

Index

- Hiding the seams
- Uses PHP templates
- The MVC Pattern
- MVC Layering

1. Hiding the seams

- Although this distinction between file types may be useful to you, the developer, there`s no reason for users to know which kind of technology has been used.
- These days, professional developers place a lot of importance on the URLs they put out into the world.
- In general, URLs should be as permanent as possible.

1. Hiding the seams

- An easy way to eliminate filename extensions in your URLs is to take advantage of directory indexes:
  - When a URL points at a directory on your web server, instead of a particular file, the web server will look for a file named index.html or index.php inside that directory, and display that file in response to the request.

1. Hiding the seams

A.3.10. Follow these steps:

- Create a new folder called today.
- Inside create a file called index.php, with this code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Today&rsquo;s Date</title>
  </head>
  <body>
    <p>Today&rsquo;s date (according to this web server) is
      <?php

      echo date('l, F jS Y.');

      ?>
    </p>
  </body>
</html>
```

2. Use PHP Templates

- As the amount of PHP code that goes into generating your average page grows, however, maintaining this mixture of HTML and PHP code can become unmanageable.
- It's far too easy for designers to accidentally modify the PHP code, causing errors.

2. Use PHP Templates

- A much more robust approach is to separate out the bulk of your PHP code so that it resides in its own file, leaving the HTML largely unpolluted by PHP code.
- <u>The key to doing this is the PHP **include** statement</u>!
  - You can insert the contents of another file into your PHP code at the point of the statement.

2. Use PHP Templates

- Example:

```
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Counting to Ten</title>
  </head>
  <body>
    <p>
      <?php
      for ($count = 1; $count <= 10; ++$count)
      {
        echo "$count ";
      }
      ?>
    </p>
  </body>
</html>
```

count10.php

2. Use PHP Templates

index.php

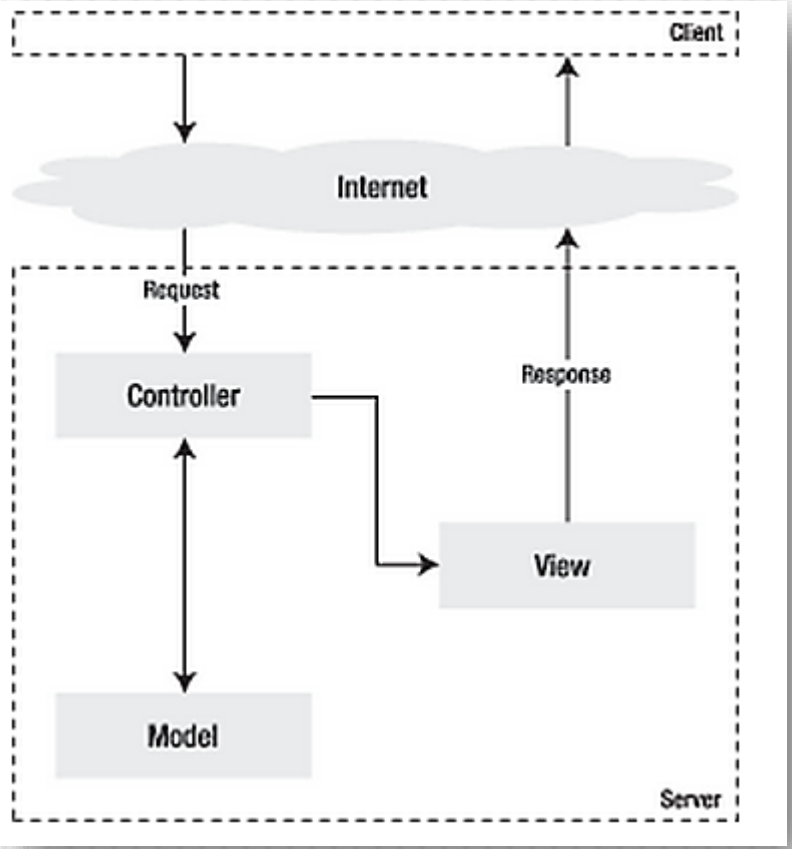
```php
<?php
$output = '';
for ($count = 1; $count <= 10; ++$count)
{
    $output .= "$count ";
}

include 'count.html.php';
```

count.html.php

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Counting to Ten</title>
  </head>
  <body>
    <p>
      <?php echo $output; ?>
    </p>
  </body>
</html>
```

3. The MVC Pattern

- MVC architecture, which consists of three levels:
  - The **model** represents the information on which the application operates (its business logic).
  - The **view** renders the model into a web page suitable for interaction with the user.
  - The **controller** responds to user actions and invokes changes on the model or view as appropriate.



3. The MVC Pattern

- A PHP script that responds to a browser request by selecting one of several PHP templates to fill in and send back is commonly called a controller.
- A controller contains the logic that controls which template is sent to the browser.

4. MVC Layering

- Flat Programming:

Estructura de tabla

#	Nombre	Tipo
1	id	int(11)
2	created_at	varchar(20)
3	title	varchar(20)
4	body	varchar(50)

```php
<?php
// index.php
$link = new PDO("mysql:host=localhost;dbname=blog_db", 'myuser', 'mypassword');

$result = $link->query('SELECT id, title FROM post');
?>

<!DOCTYPE html>
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php while ($row = $result->fetch(PDO::FETCH_ASSOC)): ?>
            <li>
                <a href="/show.php?id=<?= $row['id'] ?>">
                    <?= $row['title'] ?>
                </a>
            </li>
            <?php endwhile ?>
        </ul>
    </body>
</html>

<?php
$link = null;
?>
```

- No error checking
- Poor organization
- Difficult to reuse the code

4. MVC Layering

1. Isolating the Presentation:
   - Split into two parts:
     A. The pure PHP code with all the business logic goes in a controller script:

```php
// index.php
$link = new PDO("mysql:host=localhost;dbname=blog_db", 'myuser', 'mypassword');

$result = $link->query('SELECT id, title FROM post');

$posts = array();
while ($row = $result->fetch(PDO::FETCH_ASSOC)) {
    $posts[] = $row;
}

$link = null;

// include the HTML presentation code
require 'templates/list.php';
```

index.php

4. MVC Layering

B. The HTML code, containing template-like PHP syntax, is stored in a view script:

```
1  <!-- templates/list.php -->
2  <!DOCTYPE html>
3  <html>
4      <head>
5          <title>List of Posts</title>
6      </head>
7      <body>
8          <h1>List of Posts</h1>
9          <ul>
10             <?php foreach ($posts as $post): ?>
11             <li>
12                 <a href="/show.php?id=<?= $post['id'] ?>">
13                     <?= $post['title'] ?>
14                 </a>
15             </li>
16             <?php endforeach ?>
17         </ul>
18     </body>
19 </html>
```

list.php
(also list.html.php)

4. MVC Layering

- A good rule of thumb to determine whether the view is clean enough is that it should contain only a minimum amount of PHP code, in order to be understood by an HTML designer without PHP knowledge.
- The most common statements in views are echo, if, foreach, and that's about all.
- Also, there should not be PHP code echoing HTML tags.
- So far the application contains only one page. **But what if a second page needed to use the same database connection, or even the same array of blog posts? …**

4. MVC Layering

2. Isolating the Application (Domain) Logic:

```php
// model.php
function open_database_connection()
{
    $link = new PDO("mysql:host=localhost;dbname=blog_db", 'myuser', 'mypassword');

    return $link;
}

function close_database_connection(&$link)
{
    $link = null;
}

function get_all_posts()
{
    $link = open_database_connection();

    $result = $link->query('SELECT id, title FROM post');

    $posts = array();
    while ($row = $result->fetch(PDO::FETCH_ASSOC)) {
        $posts[] = $row;
    }
    close_database_connection($link);

    return $posts;
}
```

model.php

4. MVC Layering


```php
// index.php
require_once 'model.php';

$posts = get_all_posts();

require 'templates/list.php';
```

index.php

Find the difference between *include* and *require*. Moreover, find the difference between them and *include_once*/*require_once*.

4. MVC Layering

3. Isolating the Layout:
   - The only part of the code that *can't* be reused is the page layout.
   - Fix that by creating a new *templates/layout.php* file:

```
<!-- templates/layout.php -->
<!DOCTYPE html>
<html>
    <head>
        <title><?= $title ?></title>
    </head>
    <body>
        <?= $content ?>
    </body>
</html>
```

4. MVC Layering


```php
<!-- templates/list.php -->
<?php $title = 'List of Posts' ?>

<?php ob_start() ?>
    <h1>List of Posts</h1>
    <ul>
        <?php foreach ($posts as $post): ?>
        <li>
            <a href="/show.php?id=<?= $post['id'] ?>">
                <?= $post['title'] ?>
            </a>
        </li>
        <?php endforeach ?>
    </ul>
<?php $content = ob_get_clean() ?>

<?php include 'layout.php' ?>
```

This process is called Templating

4. MVC Layering

4. Adding a Blog "show" Page:
   - To begin, create a new function in the model.php file that retrieves an individual blog result based on a given id:

```php
// model.php
function get_post_by_id($id)
{
    $link = open_database_connection();

    $query = 'SELECT created_at, title, body FROM post WHERE  id=:id';
    $statement = $link->prepare($query);
    $statement->bindValue(':id', $id, PDO::PARAM_INT);
    $statement->execute();

    $row = $statement->fetch(PDO::FETCH_ASSOC);

    close_database_connection($link);

    return $row;
}
```

4. MVC Layering

- Next, create a new file called show.php - the controller for this new page:

```
1  // show.php
2  require_once 'model.php';
3
4  $post = get_post_by_id($_GET['id']);
5
6  require 'templates/show.php';
```

4. MVC Layering

- Finally, create the new template file - templates/show.php - to render the individual blog post:

```
1   <!-- templates/show.php -->
2   <?php $title = $post['title'] ?>
3
4   <?php ob_start() ?>
5       <h1><?= $post['title'] ?></h1>
6
7       <div class="date"><?= $post['created_at'] ?></div>
8       <div class="body">
9           <?= $post['body'] ?>
10      </div>
11  <?php $content = ob_get_clean() ?>
12
13  <?php include 'layout.php' ?>
```

4. MVC Layering

- Problems we can find in previous example:
  1. A missing or invalid id query parameter will cause the page to **crash**. It would be better if this caused a **404** page to be rendered, but this can't really be done easily yet…
  2. **Another major problem is that each individual controller file must include the model.php file**. What if each controller file suddenly needed to include an additional global task (for instance, some change related with security)? As it stands now, that code would need to be added to every controller file.

4. MVC Layering

5. The solution is to introduce a "Front Controller":
   - With one file handling all requests, you can centralize things such as security handling, configuration loading, and routing.
   - In this application, *index.php* must now be smart enough to render the blog post list page *or* the blog post show page based on the requested URI…

4. MVC Layering


```
2   require_once 'model.php';
3   require_once 'controllers.php';
4   define('ROOT','/lesson3/Through_MVC/fromFlat/');
5
6   // route the request internally
7   $uri = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);
8   if (ROOT === $uri) {
9       list_action();
10  } elseif ((ROOT.'index.php/show') === $uri && isset($_GET['id'])) {
11      show_action($_GET['id']);
12  } else {
13      header('HTTP/1.1 404 Not Found');
14      echo '<html><body><h1>Page Not Found</h1></body></html>';
15
16  }
17
```

4. MVC Layering

- For organization, both controllers (formerly *index.php and show.php*) are now PHP functions and each has been moved into a separate file named *controllers.php*:

```php
// controllers.php
function list_action()
{
    $posts = get_all_posts();
    require 'templates/list.php';
}

function show_action($id)
{
    $post = get_post_by_id($id);
    require 'templates/show.php';
}
```

- You will need also to change the href inside of list view in order to point to:

```php
<a href="./index.php/show?id=<?= $post['id'] ?>">
```

4. MVC Layering

- Another advantage of a front controller is flexible URLs. Notice that the URL to the blog post show page could be changed from /show to /read by changing code in only one location.