

# The Dispatcher

*An alternative way to check login states in multiple restricted URLs*

Deniz DIZMAN, 2007

[deniz.dizman@gmail.com](mailto:deniz.dizman@gmail.com)

<http://dendiz.blogspot.com>

## Abstract

This paper describes and discusses a way to bypass user login state for all URL requests checking in projects with multiple restricted URLs.

## Implementation

Nowadays most of the websites provide a way for the user to customize their view of the site. Its also very common for users to have secured/private data on theirs accounts. Due to this demand, websites have developed login/authentication systems, and integrated them into their applications. User authentication can be done on web server level for example using the apache `.htaccess` control system, or it can be done at application level using sessions and cookies. Due to the nature of the web, users can always type in an URL to the address bar of their browser and request a specific address, that would otherwise be requested through links within the application. This brings up the opportunity for the user to actually request restricted URLs directly. Therefore if application level authentication checking is used, every restricted URL should have a login check code at the very beginning of the code that executes, and should warn the user to login before he/she can access this site. This is a typical case of redundant code, which should be avoided at all costs. It also has the disadvantage of being forgotten, which can lead to serious security flaws within the application. Lets examine this in more detail with an example:

For this example we will have the following directory structure:

`/index.php`: the main page

`/authenticate.php`: the authentication page

`/restricted.php`: the restricted area, which requires a successful login to reach.

The flow we want is going to be something like this:

The user will request `index.php`, and if the user has logged in successfully, `index.php` will redirect this user to `restricted.php`. If the user hasn't logged in `index.php` will redirect the user to `authenticate.php`. When the user logs in from `authenticate.php`, he/she will be redirected to `restricted.php`. Another constraint is if the user requests `restricted.php` directly from the URL, there should be a checking mechanism that should restrict access if the user hasn't logged in and redirect him/her to the login page. Now lets take a look at the codes to make things clearer:

`index.php`:

```
<?php
    if ($_SESSION["loggedin"] == true)
        redirect("/restricted.php");
    else
        redirect("/authenticate.php");
?>
```

Quite straight forward 4 lines of code. Lets assume there is a function called `redirect()`, which just spits out some HTTP headers, and does the redirection stuff.

authenticate.php:

```
<?php
    if ($_POST["password"] == "myweakpassword") {
        $_SESSION["loggedin"] = true;
        redirect("/restricted.php");
    } else {
        $_SESSION["loggedin"] = false;
        echo "bad login";
    }
?>
```

We'll assume here the login form sends the password in a field called "password" just for simplicity sake.

restricted.php:

```
<?php
    if ($_SESSION["loggedin"] == true) {
        echo "you are at restricted grounds.";
    } else {
        redirect("/authenticate.php");
    }
?>
```

Again quite a simple piece of code, nothing much to explain here.

What should strike the quality coders eye and bother him here is the constant checking for the authentication state using `if()`'s. These `if()`'s are redundant code, and tend to be forgotten. Image forgetting the login check in the `restricted.php` file! The user would have direct access to a place that requires credentials. I thing you got the idea here.

Now let me introduce a simple method to overcome this. I used this method in a community site project I was building. I solved this problem quite well but it had its drawbacks. The system is quite primitive actually but I will explain how to enhance it later.

The main building block of this methods involves all of the request going to `index.php`.

So all direct access to other URLs are blocked. The user makes a call like

`index.php?page=restricted` or `index.php?page=authentication`. There should be switch-case clause in `index.php` that makes the decisions on which page is going to be displayed. Lets see some code:

index.php:

```
<?php
    if($_SESSION["loggedin"] == false)
    {
        require_once("/out.of.reach/authenticate.php");
        exit;
    }
    switch($_GET["page"]) {
        case "authenticate":
            require_once("/out.of.reach/authenticate.php"); break;
        case "restricted": require_once("/out.of.reach/restricted.php"); break;
        default: echo "invalid page"; break;
    }
?>
```

We do some login checking, if the user has not logged in already, we include the authentication page. If the user has logged in already, we just include the page the user requested.

```

authenticate.php:
<?php
    if ($_POST["password"] == "myweakpassword") {
        $_SESSION["loggedin"] = true;
        redirect("/index.php?page=restricted");
    } else {
        $_SESSION["loggedin"] = false;
        echo "bad login";
    }
?>
not much change here.

```

```

restricted.php:
<?php
    echo "restricted grounds.";
?>

```

As you can see we don't need any session checks here, because it's all done in `index.php`. One main important point here is that the required files must not be open to web access, otherwise you're just doing security through obscurity, which is a bad practice. Though this seems like a clean approach, the code gets horribly confusing as the project size increases. Imagine having 100+ cases in the switch-case statement! This was the draw back I mentioned earlier.

Now for the real juicy part: We'll add some advanced techniques like *dynamic class loading* to this approach. Dynamic class loading is on the fly object creation according to parameters by the user. This method requires that you use an object oriented approach. The user will pass on the required class name and method from the URL as parameters and the dispatcher will dynamically create the necessary class and invoke the requested method. We'll modify our `index.php` to read something like this:

```

<?php
    //do the classical logged-in check.
    //do some error checking for empty parameters.
    require_once($_GET["object"]."class.php"); //java like naming convention
    $o = new $_GET["object"];
    $o->$_GET["method"];
?>

```

I assume that you followed a java like naming convention for class files where the file name ends in class name followed by a `.class.php`. These dynamically required classes generate the page requested by the user. This is the classical approach used in the MVC (Model View Controller) for the controllers. Lets take a look at the `restricted.php` file:

```

<?php
class Restricted {
    function __construct() {}
    function display() {
        echo "restricted grounds";
    }
}
?>

```

When we make a call like `index.php?object=Restricted&method=display` the requested file will be automatically included, the class will be instantiated and the method will be called. Once again the session control is only done once, which was our main goal. And we get a nice side product: A very clean dispatcher on the contrary to the letting all the calls run through the switch-case statement in the `index.php` file. This also provides a very nice basis for caching the generated PHP output for building accelerators.