PHP LOGIN AND AUTHENTICATION:

The Complete Tutorial (2019)

Today you will learn exactly how to build a **PHP login and authentication** class.

In fact, this step-by-step guide will show you:

- How to store the accounts on the database
- How to use Sessions and MySQL for login
- All you need to know about security
- ...and more

So: if you need to work on a PHP login system, this is the guide you are looking for.

Let's get started.

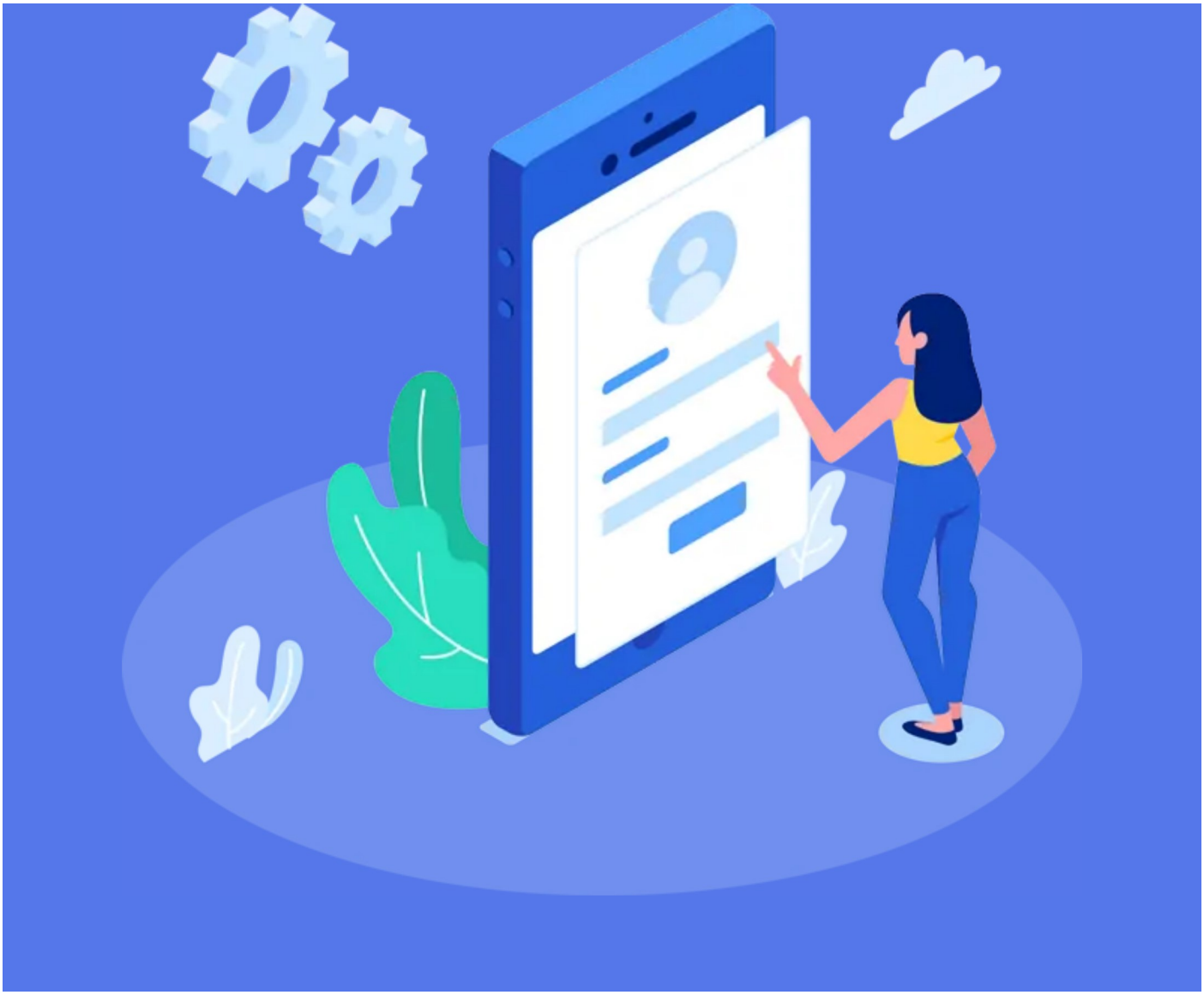


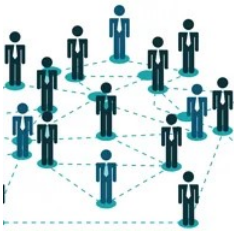
TABLE OF CONTENTS

STEP 1

Getting Started

STEP 2

Database Structure



STEP 3

Add, Edit And Delete Acco

STEP 4

Login And Logout

STEP 5

Examples and Download

STEP 6

Login Security

STEP 7

Questions & Answers

STEP 5

STEP 6

Examples and Download

STEP 7

Questions & Answers

Login Security

Step 1:

Getting Started

In this chapter you will learn how your authentication system is structured.

You will also see how to connect to the database and how to start the PHP Session.

It will only take a few minutes.

Let's go.

In this tutorial you are going to build a basic **PHP login and authentication system**.

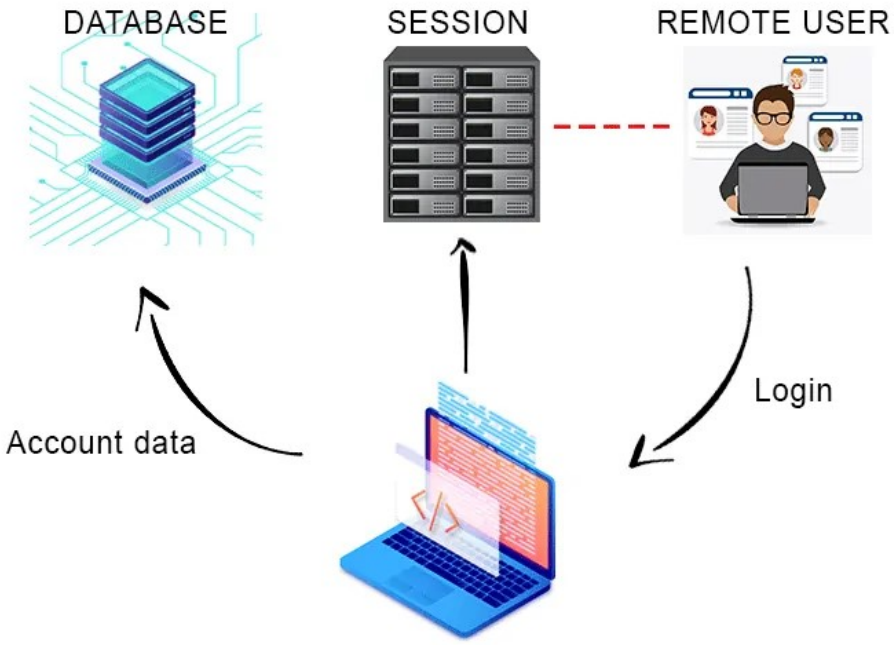
This system is composed of three different parts:

- A **Database** where to store the accounts information. All the details are in the next chapter.

- A **PHP Class** where to write the code logic for all the operations (add an account, login and logout...). This class will be called *"Account"*.

- A way to **remember** the remote clients that have already been authenticated. For this purpose, you're

going to use PHP Sessions.



So far, so good.

First of all, let's create an example script where you will use the *Account* class.

Open your favourite code editor, create an empty PHP script an save it as *"myApp.php"* inside a directory of your choice.

> **Note:** you need a working local PHP development environment (like XAMPP).
>
> If you need help setting up one, read the Getting Started chapter of my "How to learn PHP" tutorial.

Next, you need to connect to your SQL database.

The best way to do it is to create a separate “include” file with the connection code, like this one taken from my MySQL tutorial:

```
<?php

/* Host name of the MySQL server */
$host = 'localhost';

/* MySQL account username */
$user = 'myUser';

/* MySQL account password */
$passwd = 'myPasswd';

/* The schema you want to use */
$schema = 'mySchema';

/* The PDO object */
$pdo = NULL;

/* Connection string, or "data source name" */
$dsn = 'mysql:host=' . $host . ';dbname=' . $schema;

/* Connection inside a try/catch block */
try
{
   /* PDO object creation */
   $pdo = new PDO($dsn, $user,  $passwd);

   /* Enable exceptions on errors */
   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EX
}
catch (PDOException $e)
{
   /* If there is an error an exception is thrown */
   echo 'Database connection failed.';
   die();
}
```

Change the connection parameters as required, then save the above code as a PHP script named *“db_inc.php”*

inside the same directory of *myApp.php*.

Every time you need to use the database, simply include this file and the **$pdo** connection object will be available as a global variable.

So, go back to *myApp.php* and include the database connection file:

```
1  <?php
2
3  require './db_inc.php';
4
```

Very handy, isn't it?

Now, create one more PHP script which will contain the **Account class** (for now just leave it empty, you'll go back to it later). Save it as *"account_class.php"* inside the same directory.

Just like for *db_inc.php*, you can include this script every time you will need to use the Account class in any of your applications.

Let's do that right away in *myApp.php*:

```
1  <?php
2
3  require './db_inc.php';
4  require './account_class.php';
5
```

Here you go.

The last thing to do is to **start the PHP Session**. This is easily done with the *session_start()* function.

session_start() must be called before any output is sent to the remote client, therefore it's a good practice to call it before including other scripts.

You can do this in *myApp.php* just like this:

```
1  <?php
2
3  session_start();
4
5  require './db_inc.php';
6  require './account_class.php';
7
```

Very well!

Now let's dive into the details, starting from the database structure.

Step 2:

Database Structure

This chapter will show you exactly how to set up the database tables used by the Account class.

You'll get the step-by-step instructions to create the tables yourself, including the full SQL code.

You'll complete this step in no time.

Let's begin.

Now, you're going to create the database tables where the accounts data is stored.

If you're not familiar with databases or if you don't know exactly how to use PHP with MySQL, you can find everything you need here:

- How to use PHP with MySQL: the complete guide

The Account class uses two MySQL tables:

- **accounts**

- **account_sessions**

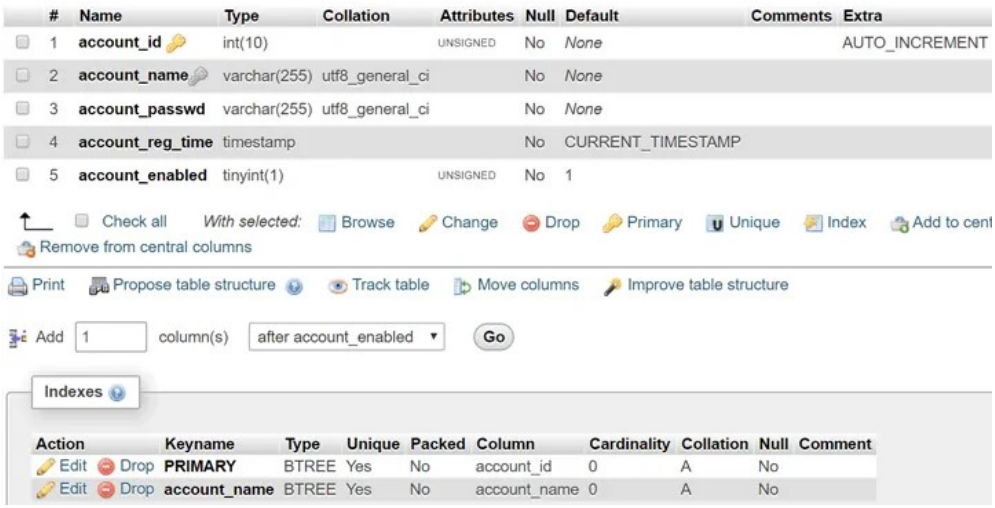
The *accounts* table contains all the registered accounts along with some basic information: username, password hash, registration time and status (enabled or disabled).

The specific table columns are:

- **account_id**: the unique identifier of the account (this is the table's *primary key*)
- **account_name**: the name used for logging in, or simply "username" (each name must be unique inside the table)
- **account_passwd**: the password hash, created with password_hash()
- **account_reg_time**: the registration timestamp (when the account has been created)
- **account_enabled**: whether the account is enabled or disabled, useful for disabling the account without deleting it from the database

Note: For this tutorial, I assume the MySQL Schema is named *mySchema*. If you are using a different schema name, be sure to change all the examples' code accordingly (just search for "mySchema" and replace it with your own).

This is how this table looks in phpMyAdmin:



#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	account_id	int(10)		UNSIGNED	No	None		AUTO_INCREMENT
2	account_name	varchar(255)	utf8_general_ci		No	None		
3	account_passwd	varchar(255)	utf8_general_ci		No	None		
4	account_reg_time	timestamp			No	CURRENT_TIMESTAMP		
5	account_enabled	tinyint(1)		UNSIGNED	No	1		

Action	Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
Edit Drop	PRIMARY	BTREE	Yes	No	account_id	0	A	No	
Edit Drop	account_name	BTREE	Yes	No	account_name	0	A	No	

As you can see, it's a very simple table.

(In the next chapters you will see exactly how each column is used.)

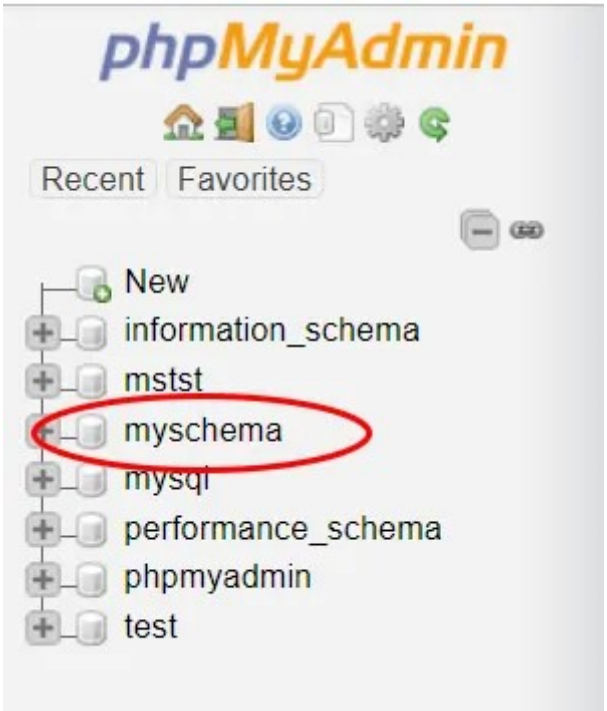
Here is the SQL code to create the table including the indexes:

```
--
-- Table structure for table `accounts`
--

CREATE TABLE `accounts` (
  `account_id` int(10) UNSIGNED NOT NULL,
  `account_name` varchar(255) NOT NULL,
  `account_passwd` varchar(255) NOT NULL,
  `account_reg_time` timestamp NOT NULL DEFAULT CURRENT_TIMES
  `account_enabled` tinyint(1) UNSIGNED NOT NULL DEFAULT '1'
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

--
-- Indexes for table `accounts`
--
ALTER TABLE `accounts`
  ADD PRIMARY KEY (`account_id`),
  ADD UNIQUE KEY `account_name` (`account_name`);
```

```
--
-- AUTO_INCREMENT for table `accounts`
--
ALTER TABLE `accounts`
  MODIFY `account_id` int(10) UNSIGNED NOT NULL AUTO_INCREMEN
```

To create the table with **phpMyAdmin**, first select your Schema from the list on the left, then click on the SQL tab and paste the code:

1 – Select your Schema from the left menu:



2 – Click on the “SQL” tab in the top menu:

3 – Paste the SQL code in the text field:

4 – Click “Go” in the bottom right corner:

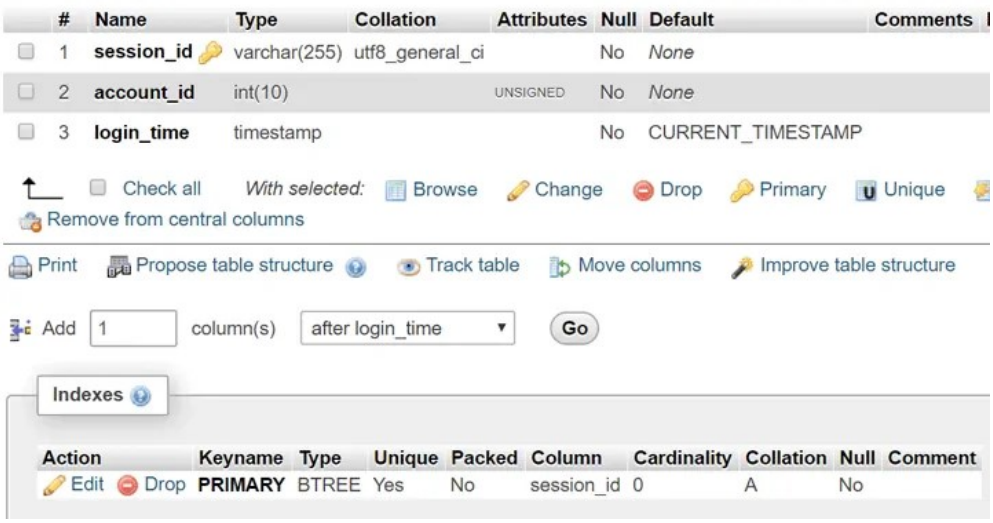
The *account_sessions* table contains the **Session IDs** used for the Session-based authentication.

Here are the table columns:

- **session_id**: the PHP Session ID of the authenticated client (this is also the *primary key*)

- **account_id**: the ID of the account (pointing to the *account_id* column of the *accounts* table)
- **login_time**: the timestamp of the session login (useful to handle session timeouts)

This is how this table looks in PhpMyAdmin:



And here is the SQL code to create the table:

```
--
-- Table structure for table `account_sessions`
--

CREATE TABLE `account_sessions` (
  `session_id` varchar(255) NOT NULL,
  `account_id` int(10) UNSIGNED NOT NULL,
  `login_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

--
-- Indexes for table `account_sessions`
--
ALTER TABLE `account_sessions`
```

```
    ADD PRIMARY KEY (`session_id`);
```

Create the *account_sessions* table by following the same steps as for the *accounts* table.

That's it!

You just finished setting up your database.

Let's move on to the next chapter (now the real fun begins....)

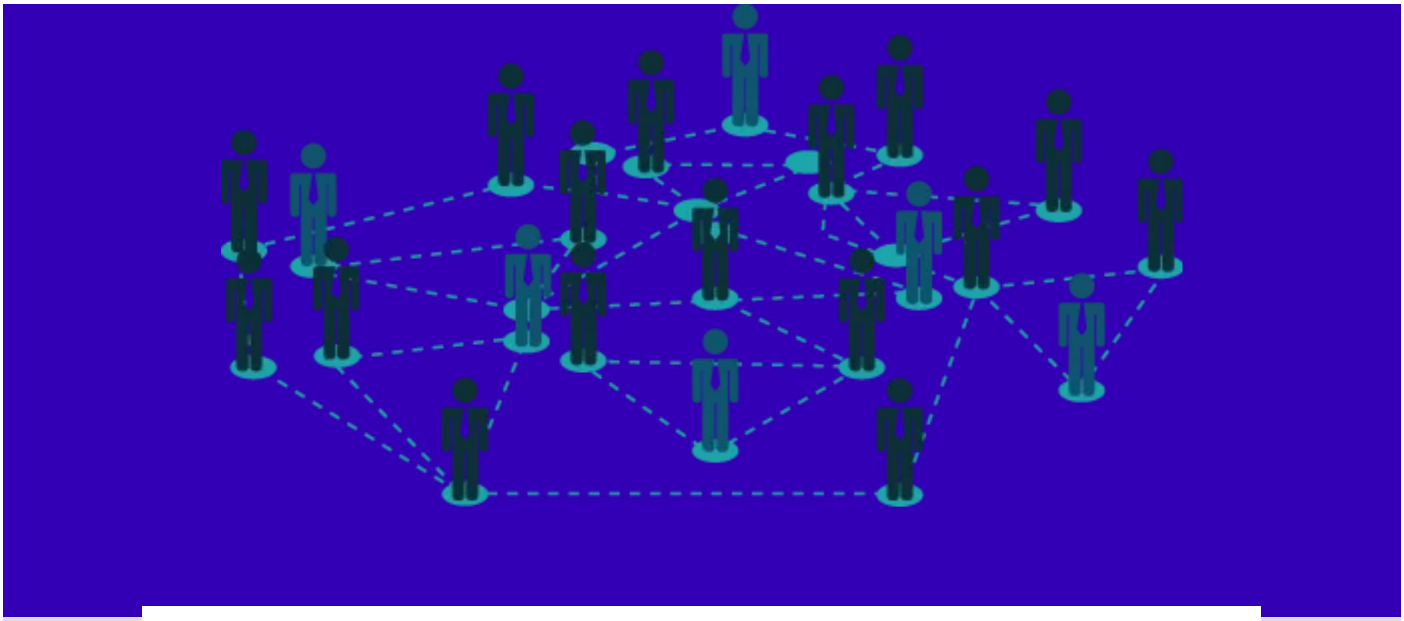
Step 3:

Add, Edit And Delete Accounts

Now you will learn exactly how to handle your accounts.

You are going to implement the class methods for adding new accounts and for editing and deleting them.

Let's dive in.



Now, it's time to write the **Account class**.

Open the *account_class.php* script you saved earlier and write the basic class structure:

```php
<?php

class Account
{
    /* Class properties (variables) */

    /* The ID of the logged in account (or NULL if there
    private $id;

    /* The name of the logged in account (or NULL if the
    private $name;

    /* TRUE if the user is authenticated, FALSE otherwis
    private $authenticated;


    /* Public class methods (functions) */

    /* Constructor */
    public function __construct()
    {
        /* Initialize the $id and $name variables to NUL
        $this->id = NULL;
        $this->name = NULL;
```

```
25          $this->authenticated = FALSE;
26      }
27
28      /* Destructor */
29      public function __destruct()
30      {
31
32      }
33  }
34
```

For now, there are just the *constructor*, the *destructor* and two class properties (which will contain the account ID, the account name and the *authenticated* flag set to TRUE after a successful login, as you will see in the next chapter).

Before moving on, let's see how **errors** are handled.

Many different errors can occur (a username is not valid, a database query failed...), and each class method must be able to signal such errors to the caller.

This is done with Exceptions.

If you never used exceptions before, don't worry: they are easier than you think.

And you will see exactly how to use them with the next examples.

All right.

Let's get started with the first class method: **addAccount()**.

How To Add A New Account

This function adds a new account to the database.

It returns the new **account ID** (that is, the *account_id* value from the *accounts* table) on success. If the operation fails, it throws an exception with a specific error message.

Let's look at the code:

```
1  /* Add a new account to the system and return its ID (th
2  public function addAccount(string $name, string $passwd)
3  {
4      /* Global $pdo object */
5      global $pdo;
6
7      /* Trim the strings to remove extra spaces */
8      $name = trim($name);
9      $passwd = trim($passwd);
10
11     /* Check if the user name is valid. If not, throw an
12     if (!$this->isNameValid($name))
13     {
14         throw new Exception('Invalid user name');
15     }
16
17     /* Check if the password is valid. If not, throw an
18     if (!$this->isPasswdValid($passwd))
19     {
```

```
            throw new Exception('Invalid password');
        }

        /* Check if an account having the same name already
        if (!is_null($this->getIdFromName($name)))
        {
            throw new Exception('User name not available');
        }

        /* Finally, add the new account */

        /* Insert query template */
        $query = 'INSERT INTO myschema.accounts (account_nam

        /* Password hash */
        $hash = password_hash($passwd, PASSWORD_DEFAULT);

        /* Values array for PDO */
        $values = array(':name' => $name, ':passwd' => $hash

        /* Execute the query */
        try
        {
            $res = $pdo->prepare($query);
            $res->execute($values);
        }
        catch (PDOException $e)
        {
           /* If there is a PDO exception, throw a standard
           throw new Exception('Database query error');
        }

        /* Return the new ID */
        return $pdo->lastInsertId();
    }

```

Let me explain all the steps.

Before adding the new account to the database, *addAccount()* performs some checks on the input variables to make sure they are correct.

In particular:

- it calls **isNameValid()**, a class method that returns TRUE if the passed string can be a valid username
- then, it calls **isPasswdValid()**, that returns TRUE if the passed string can be a valid password
- finally, it calls **getIdFromName()**: this method looks for the passed username on the database, and it returns its *account_id* if it exists or NULL if it doesn't

If the username passed to addAccount() is not valid (for example, it's too short), an exception is thrown and the method stops its execution.

Similarly, an exception is thrown if the password is not valid or if the username is not available.

If everything is fine, the new account is finally added to the database and **its ID is returned**.

(An exception is also thrown if a PDO exception is caught, meaning there was an error while executing the SQL query).

Suggested reading: **How to hash passwords in PHP**

Here is the full code of *isNameValid()*, *isPasswdValid()* and *getIdFromName()*:

```php
/* A sanitization check for the account username */
public function isNameValid(string $name): bool
{
    /* Initialize the return variable */
    $valid = TRUE;

    /* Example check: the length must be between 8 and 1
    $len = mb_strlen($name);

    if (($len < 8) || ($len > 16))
    {
        $valid = FALSE;
    }

    /* You can add more checks here */

    return $valid;
}

/* A sanitization check for the account password */
public function isPasswdValid(string $passwd): bool
{
    /* Initialize the return variable */
    $valid = TRUE;

    /* Example check: the length must be between 8 and 1
    $len = mb_strlen($passwd);

    if (($len < 8) || ($len > 16))
    {
        $valid = FALSE;
    }

    /* You can add more checks here */

    return $valid;
}

/* Returns the account id having $name as name, or NULL
public function getIdFromName(string $name): ?int
{
    /* Global $pdo object */
    global $pdo;

    /* Since this method is public, we check $name again
    if (!$this->isNameValid($name))
    {
        throw new Exception('Invalid user name');
```

```
    }

    /* Initialize the return value. If no account is fou
    $id = NULL;

    /* Search the ID on the database */
    $query = 'SELECT account_id FROM myschema.accounts W
    $values = array(':name' => $name);

    try
    {
        $res = $pdo->prepare($query);
        $res->execute($values);
    }
    catch (PDOException $e)
    {
       /* If there is a PDO exception, throw a standard
       throw new Exception('Database query error');
    }

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

    /* There is a result: get it's ID */
    if (is_array($row))
    {
        $id = intval($row['account_id'], 10);
    }

    return $id;
}

```

Both *isNameValid()* and *isPasswdValid()* perform a simple length check.

However, you can easily edit them to perform additional checks, for example forcing the password to have at least one capital letter and one digit.

If you have any doubt about these methods, just leave me a comment.

Ok, BUT:

How do you use this method from your application?

Well, it's very simple.

This is what you need to write inside *myApp.php*:

```
<?php

session_start();

require './db_inc.php';
require './account_class.php';

$account = new Account();

try
{
    $newId = $account->addAccount('myNewName', 'myPasswo
}
catch (Exception $e)
{
    /* Something went wrong: echo the exception message
    echo $e->getMessage();
    die();
}

echo 'The new account ID is ' . $newId;

```

It's easy to read, elegant and efficient.

All right:

Let's move on to the next method: **editAccount()**.

How To Edit An Account

This method lets you change the name, the password or the status (enabled or disabled) of a specific account.

The first function argument is the **Account ID** of the account to be changed.

The next arguments are the new values to be set: the new name, the new password and the new status (as a Boolean value).

Like *addAccount()*, this function checks for the validity of the parameters before actually modifying the account on the database.

The name and the password are verified with *isNameValid()* and *isPasswdValid()*, the same methods used by *addAccount()*.

The account ID is verified by a new method: **isIdValid()**.

Also, the new name must not be already by other accounts.

Here's the code:

```
/* Edit an account (selected by its ID). The name, the p
public function editAccount(int $id, string $name, strin
{
    /* Global $pdo object */
    global $pdo;

    /* Trim the strings to remove extra spaces */
    $name = trim($name);
    $passwd = trim($passwd);

    /* Check if the ID is valid */
    if (!$this->isIdValid($id))
    {
        throw new Exception('Invalid account ID');
    }

    /* Check if the user name is valid. */
    if (!$this->isNameValid($name))
    {
        throw new Exception('Invalid user name');
    }

    /* Check if the password is valid. */
    if (!$this->isPasswdValid($passwd))
    {
        throw new Exception('Invalid password');
    }

    /* Check if an account having the same name already
    $idFromName = $this->getIdFromName($name);

    if (!is_null($idFromName) && ($idFromName != $id))
    {
        throw new Exception('User name already used');
    }

    /* Finally, edit the account */

```

```
        /* Edit query template */
        $query = 'UPDATE myschema.accounts SET account_name

        /* Password hash */
        $hash = password_hash($passwd, PASSWORD_DEFAULT);

        /* Int value for the $enabled variable (0 = false, 1
        $intEnabled = $enabled ? 1 : 0;

        /* Values array for PDO */
        $values = array(':name' => $name, ':passwd' => $hash

        /* Execute the query */
        try
        {
            $res = $pdo->prepare($query);
            $res->execute($values);
        }
        catch (PDOException $e)
        {
           /* If there is a PDO exception, throw a standard
           throw new Exception('Database query error');
        }
    }

```

And here is the code of the *isIdValid()* method:

```
/* A sanitization check for the account ID */
public function isIdValid(int $id): bool
{
    /* Initialize the return variable */
    $valid = TRUE;

    /* Example check: the ID must be between 1 and 10000

    if (($id < 1) || ($id > 1000000))
    {
        $valid = FALSE;
    }

    /* You can add more checks here */

```

```
16      return $valid;
17  }
18
```

The last method of this chapter is **deleteAccount()**.

This is quite straightforward: it takes an *account ID* and deletes it. The account Sessions inside the *account_sessions* table are also deleted.

Here's the code:

```
1   /* Delete an account (selected by its ID) */
2   public function deleteAccount(int $id)
3   {
4       /* Global $pdo object */
5       global $pdo;
6
7       /* Check if the ID is valid */
8       if (!$this->isIdValid($id))
9       {
10          throw new Exception('Invalid account ID');
11      }
12
13      /* Query template */
14      $query = 'DELETE FROM myschema.accounts WHERE accoun
15
```

```
16      /* Values array for PDO */
17      $values = array(':id' => $id);
18
19      /* Execute the query */
20      try
21      {
22          $res = $pdo->prepare($query);
23          $res->execute($values);
24      }
25      catch (PDOException $e)
26      {
27         /* If there is a PDO exception, throw a standard
28         throw new Exception('Database query error');
29      }
30
31      /* Delete the Sessions related to the account */
32      $query = 'DELETE FROM myschema.account_sessions WHER
33
34      /* Values array for PDO */
35      $values = array(':id' => $id);
36
37      /* Execute the query */
38      try
39      {
40          $res = $pdo->prepare($query);
41          $res->execute($values);
42      }
43      catch (PDOException $e)
44      {
45         /* If there is a PDO exception, throw a standard
46         throw new Exception('Database query error');
47      }
48  }
49
```

Very well!

You're ready for the next step: **login and logout**.

Step 4:

Login And Logout

In this chapter you will learn how remote clients can login (and logout) using your class.

You will see how to:

- login with username and password, and
- login using PHP Sessions

Let's go.

A remote client can login in two ways.

The first is the classic way: by providing a username and password couple. This is the "real" login.

After a successful login, a Session for that remote client is started.

The second way is by restoring a previously started Session, without the need for the client to provide name and password again.

Let's start with the username and password login.

Login With Username And Password

This is done with the **login()** class method.

Here's what this function does:

- it checks if the provided *$username* and *$password* variables are valid. If they are not, the function immediately returns FALSE (meaning the authentication request failed)
- then, in looks for the username on the database account list. If it's there, it checks the password with password_verify()
- If the password matches then the client is authenticated,

and the function sets the class properties related to the current account (its ID and its name). The *$authenticated* property is set to TRUE.

- finally, it creates or updates the client Session and returns TRUE, meaning the client has successfully authenticated.

(To learn more about password hashing and verification: **PHP password hashing tutorial**)

Here's the code:

```
/* Login with username and password */
public function login(string $name, string $passwd): boo
{
    /* Global $pdo object */
    global $pdo;

    /* Trim the strings to remove extra spaces */
    $name = trim($name);
    $passwd = trim($passwd);

    /* Check if the user name is valid. If not, return F
    if (!$this->isNameValid($name))
    {
        return FALSE;
    }

    /* Check if the password is valid. If not, return FA
    if (!$this->isPasswdValid($passwd))
    {
        return FALSE;
    }

    /* Look for the account in the db. Note: the account
    $query = 'SELECT * FROM myschema.accounts WHERE (acc

    /* Values array for PDO */
    $values = array(':name' => $name);
```

```
28
29      /* Execute the query */
30      try
31      {
32          $res = $pdo->prepare($query);
33          $res->execute($values);
34      }
35      catch (PDOException $e)
36      {
37         /* If there is a PDO exception, throw a standard
38         throw new Exception('Database query error');
39      }
40
41      $row = $res->fetch(PDO::FETCH_ASSOC);
42
43      /* If there is a result, we must check if the passwo
44      if (is_array($row))
45      {
46          if (password_verify($passwd, $row['account_passw
47          {
48              /* Authentication succeeded. Set the class p
49              $this->id = intval($row['account_id'], 10);
50              $this->name = $name;
51              $this->authenticated = TRUE;
52
53              /* Register the current Sessions on the data
54              $this->registerLoginSession();
55
56              /* Finally, Return TRUE */
57              return TRUE;
58          }
59      }
60
61      /* If we are here, it means the authentication faile
62      return FALSE;
63  }
64
```

isNameValid() and *isPasswdValid()* are the very same methods you already know.

What's new here is **registerLoginSession()**.

So, what does this function do, exactly?

It's very easy:

Once the remote client has been authenticated, this function **gets the ID of the current PHP Session** and saves it on the database together with the *account ID*.

This way, the next time the same remote client will connect, it will be automatically authenticated just by looking at its Session ID.

(The Session ID is linked to the remote browser, so it will remain the same the next time the same client will connect again).

If you are not familiar with Sessions, I suggest you spend a few minutes reading my guide:

- PHP Sessions explained

Here's the code for *registerLoginSession()*:

```
1  /* Saves the current Session ID with the account ID */
2  private function registerLoginSession()
3  {
4      /* Global $pdo object */
5      global $pdo;
6
7      /* Check that a Session has been started */
8      if (session_status() == PHP_SESSION_ACTIVE)
9      {
10         /*  Use a REPLACE statement to:
11             - insert a new row with the session id, if i
12             - update the row having the session id, if i
13         */
14         $query = 'REPLACE INTO myschema.account_sessions
```

```
15          $values = array(':sid' => session_id(), ':accoun
16
17          /* Execute the query */
18          try
19          {
20              $res = $pdo->prepare($query);
21              $res->execute($values);
22          }
23          catch (PDOException $e)
24          {
25             /* If there is a PDO exception, throw a stand
26             throw new Exception('Database query error');
27          }
28      }
29  }
30
```

Let me explain how it works.

Remember the *account_sessions* table?

registerLoginSession() **inserts a new row into that table**, with the current Session ID (given by the session_id() function) in the *session_id* column, the authenticated account ID in the *account_id* column and the *login_time* column set to the current timestamp.

Put simply, this new row says:

"This Session ID (in the session_id column) belongs to a remote user who has already logged in as the account with this ID (in the account_id column), and it has logged in at this time (the login_time column)."

If a row with the same Session ID (that is the table's primary key) already exists, that row is replaced.

If you were wondering what the "REPLACE" SQL

command does, it's exactly that, that is:

- inserts a new row if another row with the same key (the Session ID) does not exits
- otherwise, it simply updates that row

Very handy 🙂

Now let's move on to the **Session-based login**.

Session-based Login

The Session-based login is done with the **sessionLogin()** method.

This function gets the current Session ID (using *session_id()*) and looks for it into the *account_sessions* table.

If it's there, it also checks that:

- the Session is not older than 7 days
- the account linked to the session is **enabled**

If everything is ok then the client is authenticated, the account-related class properties are set, and the method returns **TRUE**.

If, for some reason, the authentication fails then the function returns **FALSE**.

Here is the *sessionLogin()* code:

```
/* Login using Sessions */
public function sessionLogin(): bool
{
    /* Global $pdo object */
    global $pdo;

    /* Check that the Session has been started */
    if (session_status() == PHP_SESSION_ACTIVE)
    {
        /*
            Query template to look for the current sessi
            The query also make sure the Session is not
        */
        $query =

        'SELECT * FROM myschema.account_sessions, mysche
        'AND (account_sessions.login_time >= (NOW() - IN
        'AND (accounts.account_enabled = 1)';

        /* Values array for PDO */
        $values = array(':sid' => session_id());

        /* Execute the query */
        try
        {
```

```
            $res = $pdo->prepare($query);
            $res->execute($values);
        }
        catch (PDOException $e)
        {
            /* If there is a PDO exception, throw a stand
            throw new Exception('Database query error');
        }

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

        if (is_array($row))
        {
            /* Authentication succeeded. Set the class p
            $this->id = intval($row['account_id'], 10);
            $this->name = $row['account_name'];
            $this->authenticated = TRUE;

            return TRUE;
        }
    }

    /* If we are here, the authentication failed */
    return FALSE;
}

```

Note

Be sure to check the **Session Cookie Lifetime** parameter in your PHP configuration (usually, the *php.ini* file). This parameter is named session.cookie_lifetime, and it specifies how many seconds a Session should be kept opened.

After that time, a Session will be closed and a new Session ID will be created, forcing the remote client with the expired Session to authenticate again with username and password.

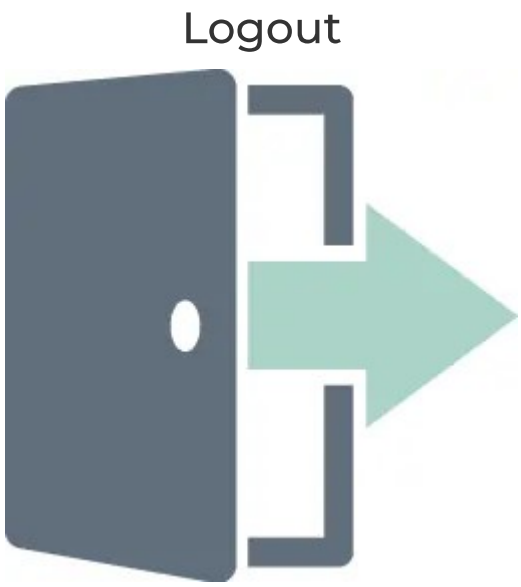
The default value is 0, meaning a Session lasts only

until the browser is closed. This is usually not very useful, so you probably want to set it higher, at least a few days.

For example, this is how to set the Session lifetime to 7 days (7 days = 604800 seconds):

session.cookie_lifetime=604800

Finally, let's see how to **logout** a remote client.



The **logout()** method clears the account-related class properties (*$id* and *$name*) and deletes the current Session from the *account_sessions* table.

The PHP Session itself is not closed because there is no need for it. Also, the Session may be needed by other sections of the web application.

However, this Session can no longer be used to log in with *sessionLogin()*.

This is the code:

```php
/* Logout the current user */
public function logout()
{
    /* Global $pdo object */
    global $pdo;

    /* If there is no logged in user, do nothing */
    if (is_null($this->id))
    {
        return;
    }

    /* Reset the account-related properties */
    $this->id = NULL;
    $this->name = NULL;
    $this->authenticated = FALSE;

    /* If there is an open Session, remove it from the a
    if (session_status() == PHP_SESSION_ACTIVE)
    {
        /* Delete query */
        $query = 'DELETE FROM myschema.account_sessions

        /* Values array for PDO */
        $values = array(':sid' => session_id());

        /* Execute the query */
        try
        {
            $res = $pdo->prepare($query);
            $res->execute($values);
        }
        catch (PDOException $e)
        {
           /* If there is a PDO exception, throw a stand
           throw new Exception('Database query error');
        }
    }
}

```

To check whether the current remote user is authenticated, you can use the **isAuthenticated()** method.

This function returns the *$authenticated* property, which is set to TRUE after a successful authentication:

```
/* "Getter" function for the $authenticated variable
    Returns TRUE if the remote user is authenticated */
public function isAuthenticated(): bool
{
    return $this->authenticated;
}

```

Hey, you are almost done!

Before looking at some **code examples** in the next chapter, there is one more *bonus method* I want to show you.

This function closes all the account open Sessions except for the current one.

In other words, it's what you need to do if you want to implement the ***"Exit from all my other open sessions"*** functionality you probably saw in some online service (like Google or Facebook).

The best part?

It's insanely simple. Here it is:

```php
/* Close all account Sessions except for the current one
public function closeOtherSessions()
{
    /* Global $pdo object */
    global $pdo;

    /* If there is no logged in user, do nothing */
    if (is_null($this->id))
    {
        return;
    }

    /* Check that a Session has been started */
    if (session_status() == PHP_SESSION_ACTIVE)
    {
        /* Delete all account Sessions with session_id d
        $query = 'DELETE FROM myschema.account_sessions

        /* Values array for PDO */
        $values = array(':sid' => session_id(), ':accoun

        /* Execute the query */
        try
        {
            $res = $pdo->prepare($query);
            $res->execute($values);
        }
        catch (PDOException $e)
        {
           /* If there is a PDO exception, throw a stand
           throw new Exception('Database query error');
        }
    }
}

```

Step 5:

Examples And Download Link

You made it!

In this chapter you will find some clear examples to better understand how to use your new Account class.

You will also find the links to download the whole PHP class file as well as the examples.

How can you use your **Account** class from your web application?

Let's see a few examples.

Go back again to your *myApp.php* example app and open it in your code editor.

Your file should look like this:

```
<?php

/* Start the PHP Session */
session_start();

/* Include the database connection file (remember to cha
require './db_inc.php';

/* Include the Account class file */
require './account_class.php';

/* Create a new Account object */
$account = new Account();

```

All right.

In the following code examples, you will see **how to perform all the Account class operations** you learned in this tutorial.

To test them yourself, copy the code snippets one at a time and paste them in *myApp.php*, after the code shown above.

After the examples, you will also find the link to download the class PHP file as well as a *myApp.php* file with all the examples shown here.

(Note: the *getId()* and *getName()* methods, used in the following examples, are simple getter functions to get the *$id* and *$name* class attributes).

Let's begin.

1. Insert a new account:

```
try
{
    $newId = $account->addAccount('myUserName', 'myPassw
}
catch (Exception $e)
{
    echo $e->getMessage();
    die();
}

echo 'The new account ID is ' . $newId;

```

2. Edit an account.

```
$accountId = 1;

try
{
    $account->editAccount($accountId, 'myNewName', 'new
}
catch (Exception $e)
{
    echo $e->getMessage();
    die();
}

echo 'Account edit successful.';

```

3. Delete an account.

```
$accountId = 1;

try
```

```
{
    $account->deleteAccount($accountId);
}
catch (Exception $e)
{
    echo $e->getMessage();
    die();
}

echo 'Account delete successful.';

```

4. Login with username and password.

```
$login = FALSE;

try
{
    $login = $account->login('myUserName', 'myPassword')
}
catch (Exception $e)
{
    echo $e->getMessage();
    die();
}

if ($login)
{
    echo 'Authentication successful.';
    echo 'Account ID: ' . $account->getId() . '<br>';
    echo 'Account name: ' . $account->getName() . '<br>'
}
else
{
    echo 'Authentication failed.';
}

```

5. Session Login.

```
$login = FALSE;

```

```php
try
{
    $login = $account->sessionLogin();
}
catch (Exception $e)
{
    echo $e->getMessage();
    die();
}

if ($login)
{
    echo 'Authentication successful.';
    echo 'Account ID: ' . $account->getId() . '<br>';
    echo 'Account name: ' . $account->getName() . '<br>'
}
else
{
    echo 'Authentication failed.';
}

```

6. Logout.

```php
try
{
    $login = $account->login('myUserName', 'myPassword')

    if ($login)
    {
        echo 'Authentication successful.';
        echo 'Account ID: ' . $account->getId() . '<br>'
        echo 'Account name: ' . $account->getName() . '<
    }
    else
    {
        echo 'Authentication failed.<br>';
    }

    $account->logout();

    $login = $account->sessionLogin();

    if ($login)
    {
```

```
        echo 'Authentication successful.';
        echo 'Account ID: ' . $account->getId() . '<br>'
        echo 'Account name: ' . $account->getName() . '<
    }
    else
    {
        echo 'Authentication failed.<br>';
    }
}
catch (Exception $e)
{
    echo $e->getMessage();
    die();
}

echo 'Logout successful.';

```

7. Close other open Sessions (if any).

```
try
{
    $login = $account->login('myUserName', 'myPassword')

    if ($login)
    {
        echo 'Authentication successful.';
        echo 'Account ID: ' . $account->getId() . '<br>'
        echo 'Account name: ' . $account->getName() . '<
    }
    else
    {
        echo 'Authentication failed.<br>';
    }

    $account->closeOtherSessions();
}
catch (Exception $e)
{
    echo $e->getMessage();
    die();
}

echo 'Sessions closed successfully.';

```

If you have any question about how to use the Account class, just leave a comment below.

Click the link below to **download a ZIP file** with:

- The Account class script (*account_class.php*)
- The *myApp.php* example file with all the above usage examples
- The *db_inc.php* file with the example PDO connection

Download The ZIP File

Would you like to talk with me and other developers about PHP and web development? Join my Facebook Group: **Alex PHP café**

See you there 🙂

Step 6:

Login Security

Security is crucial for a web application.

In this chapter you will find some quick, actionable steps

to use the Account class securely.

Here we go.

You know that **security** is crucial for web applications.

And it's even more important for a web authentication system.

So:

Let's see what are the steps you should take in order to use this class securely.

P.S.

If you want to master PHP security, you can enroll in my professional PHP Security course.

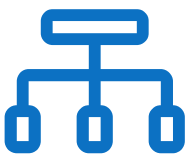
1. Ensure Password Security

This class uses password_hash() and password_verify() to store the **password hash** on the database and to match it against the plain-text password provided by the client.

These functions take care of using a **strong hashing algorithm** with a pseudo-random **salt**.

If you need to change how the accounts data is stored on your database, be sure to keep using those functions. In any case, never store the passwords in plain text and never use weak hashing algorithms (like *MD5*).

If you want to learn more about password security, go to my PHP Password Hashing tutorial.



2. Ensure Database Security

SQL Injection attacks are one of the most common attacks used against web applications.

In this tutorial, you used the PDO extension for database operation. Every potentially unsafe string is sent to the database using prepared statements.

If you are not familiar with PDO, you can use the MySQLi extension instead. MySQLi supports both prepared statements (preferred) or escaping.

In any case, I suggest you read my guide on SQL Injection prevention to make sure you know what has to be done to avoid such attacks.

If you need some clear explanation and examples on how to use PDO and MySQLi, you can find everything you need in my PHP with MySQL Complete Guide.

3. Perform Input Validation

Input validation is another cornerstone of web security.

In this class, some basic validation on the username, the password and the account ID values is done by these three methods:

- *isNameValid()*

- *isPasswdValid()*
- and *isIdValid()*

I encourage you to edit these functions and make them as strict as possible.

For example, if you decide the username must contain only a definite set of characters, a good validation step would be to check every character against a **whitelist**, like this:

```php
function whitelistText($string): bool
{
    $valid = TRUE;
    $whiteList = 'abcdefghijklmnopqrstuvwxyz123456789';

    for ($i = 0; $i < mb_strlen($string); $i++)
    {
        $char = mb_substr($string, $i, 1);

        if (mb_strpos($whiteList, $char) === FALSE)
        {
            $valid = FALSE;
        }
    }

    return $valid;
}

```

4. Ensure Sessions Security

The Session ID, used for Session-based login, is sent

inside HTTP cookies.

Therefore, the most important thing to do to make it safe is to enable **HTTPS**.

With HTTPS in place, Session attacks like *data sniffing* become much harder.

However, Sessions have other potential security flaws (like the **Session Fixation** vulnerability) that needs to be mitigated by a correct configuration.

The most important PHP configuration values you should check are Use Strict Mode, Use Only Cookies and Cookie Secure.

You can find them inside your *php.ini* file:

- *session.use_strict_mode=1*
- *session.use_only_cookies=1*
- *session.cookie_secure=1*

Make sure to leave them enabled. However, you may need to disable them when working on your local development environment.

The full list of Session security related configuration options is here.

Step 7:

Questions & Answers

In this last chapter you will find the answers to some of the most asked questions about PHP authentication.

If you have any other question, just leave a comment below.

Question #1

How do you encrypt passwords using PHP?

PHP provides an easy way to create secure password

hashes and match them against plain text passwords.

In fact, you can create a password hash with the password_hash() function, and match an existing hash against a plain text password with password_verify().

password_hash() takes care of using a strong enough hashing algorithm and adding a pseudo-random salt to the hash.

Question #2

How do you add a password salt using PHP?

A ***Salt*** is a pseudo-random string used when encrypting or hashing a string (like a password).

Salts are used to improve protection against some kinds of attack, like dictionary-based attacks.

Different algorithms use salts in different ways, but if you just want to use a salt for password safety, you can rely on the *password_hash()* function (see the previous question).

In fact, this function automatically adds a pseudo-random salt to the hash for you.

Question #3

Are PHP Session variables secure?

Session data is stored on the server where PHP is running (unless a different Session storage is used). Therefore, Session data is as secure as the server is.

Session variables are not sent through the network. Only the Session ID is.

If configured properly (see the previous chapter about **Login Security**), Sessions are safe enough for most uses.

In security-critical applications, however, it may be a good idea to set the Session timeout to a very low value (see the session.cookie_lifetime parameter).

Question #4

Can PHP Sessions be *hacked*?

Session hijacking, or *hacking*, is theoretically possible.

Two main attack types exist:

- Session Fixation, and
- Session Hijacking

Fixation attacks can be prevented by enabling Sessions Strict Mode in your PHP configuration (see the Login Security chapter for the details).

Session Hijacking attacks are a pool of different techniques for **stealing** or *predicting* a Session ID, which could then be used by the attacker to impersonate the victim.

Such attacks include traffic sniffing, **XSS** attacks and **MITM** (main-in-the-middle) attacks.

Using HTTPS mitigates or solves most of them.

Conclusion

With this tutorial you learned how a complete PHP login and authentication system works.

You saw how to perform a username/password login as well as a Session-based login.

You also learned how to add, edit and delete accounts from the database, how to be sure

your system is secure, and more.

Let me know what you think in the comments.

PS If this guide has been helpful to you, please spend a second of your time to share it... thanks!

The images used in this post have been downloaded from Freepik.