Server-side Web Development

Unit 9. Security.

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2024-25

Index

1 Security

In this unit we'll see how to apply basic security to our application. There are 2 main mechanism of security:

- **Authentication**: how the users can access the app and where their credentials are stored.
- **Authorization**: once the user has been logged in correctly, determine their **permissions** and what resources can access and which can't.

In this unit we will use the Provider App of the previous examples to show how to do all the stuff. We will use the next scripts:

- `provider_form.php`: the existent script, used to manage providers.
- `register_form.php`: a new script to auto-register users.
- `login_form.php`: a new script to log in the users.

2 Managing users

To implement the user's authentication in our app, we need two more forms, one for registering and another one for logging and a class for managing and storing logins.

2.1 Login class

The Login class represents a user, identified by their email and password. The properties are:

- id: an integer for identifying the user in the database.
- email: a string with email format.
- password: a string with at least 8 characters.

We also need a function to validate the login's data.

The code for the Login class is as follows:

```php
<?php
declare(strict_types=1);

class Login
{
    private int $id;
    private string $email;
```

```php
    private string $password;

    public function __construct()
    {
        $this->id = 0;
        $this->email = '';
        $this->password = '';
    }

    //Getters and setters ...

    public function validate(): array {
        $errors = [];
        if(empty($this->email)){
            $errors['email'] = 'Email is required';
        } elseif (!filter_var($this->email, FILTER_VALIDATE_EMAIL)){
            $errors['email'] = 'Email is not valid';
        }

        if(empty($this->password)){
            $errors['password'] = 'Password is required';
        } elseif (strlen($this->password) < 8){
            $errors['password'] = 'Password must be at least 8 characters';
        }

        return $errors;
    }
}
```

2.2 Login repository

In order to store our logins in the database, we also need a repository with the basic CRUD operations:

```php
<?php
declare(strict_types=1);
require_once __DIR__ . '/Login.php';
require_once __DIR__ . '/DBConnection.php';
require_once __DIR__ . '/IDbAccess.php';

class LoginRepository implements IDbAccess
{
```

```php
public static function getAll(): ?array
{
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        $stmt = $conn->prepare("SELECT * FROM login");
        $stmt->setFetchMode(PDO::FETCH_CLASS|PDO::FETCH_PROPS_LATE,
        ↪  'Login');
        $stmt->execute();
        return $stmt->fetchAll();
    } else {
        return null;
    }
}

public static function select($id) : ?Login
{
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        $stmt = $conn->prepare("SELECT * FROM login WHERE id = :id");
        $stmt->setFetchMode(PDO::FETCH_CLASS|PDO::FETCH_PROPS_LATE,
        ↪  'Login');
        $stmt->execute(['id' => $id]);
        $login = $stmt->fetch();
        if($login) {
            return $login;
        }
    }
    return null;
}


public static function insert($object) : false | string
{
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        $stmt = $conn->prepare("INSERT INTO login (id, email, password)
        ↪  VALUES (:id, :email, :password)");
        $stmt->execute([
            'id' => null,
            'email' => $object->getEmail(),
            'password' => $object->getPassword()
```

```
            ]);
            return $conn->lastInsertId();
        }
        return false;
    }

    public static function delete($object) : int
    {
        $conn = DBConnection::connectDB();
        if (!is_null($conn)) {
            $stmt = $conn->prepare("DELETE FROM login WHERE id = :id");
            $stmt->execute(['id' => $object->getId()]);
            return $stmt->rowCount();
        }
        return 0;
    }

    public static function update($object) : int
    {
        $conn = DBConnection::connectDB();
        if (!is_null($conn)) {
            $stmt = $conn->prepare("UPDATE login SET email = :email,
            ↪  password = :password WHERE id = :id");
            $stmt->execute([
                'id' => $object->getId(),
                'email' => $object->getEmail(),
                'password' => $object->getPassword()
            ]);
            return $stmt->rowCount();
        }
        return 0;
    }
}
```

We also need a new table in the database with the required fields:

```
CREATE TABLE `providers`.`login` (`id` INT NOT NULL AUTO_INCREMENT , `email`
↪  VARCHAR(100) NOT NULL , `password` VARCHAR(100) NOT NULL , PRIMARY KEY
↪  (`id`)) ENGINE = InnoDB;
```

The new table description:

```
DESCRIBE login;
+----------+--------------+------+-----+---------+----------------+
| Field    | Type         | Null | Key | Default | Extra          |
+----------+--------------+------+-----+---------+----------------+
| id       | int(11)      | NO   | PRI | NULL    | auto_increment |
| email    | varchar(100) | NO   |     | NULL    |                |
| password | varchar(100) | NO   |     | NULL    |                |
+----------+--------------+------+-----+---------+----------------+
```

2.3 Register

We are going to make a registration form. Make a new script called `register_form.php` with the registration code. We don't need the `id` field and only need a 'Register' button. Also, make sure that the password field is of type **password** (let the email field of type text to check the server validation):

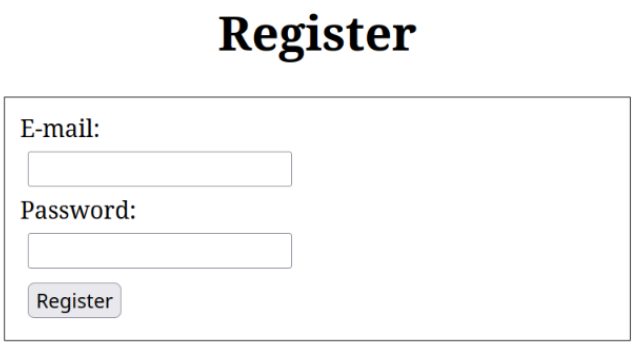

Figure 1: Register form

When the ‘Register’ button is pressed, we send the data to the same form with the POST method to do the required validations. But here we add an additional check: if the email already exists in the database, we show a new error message:

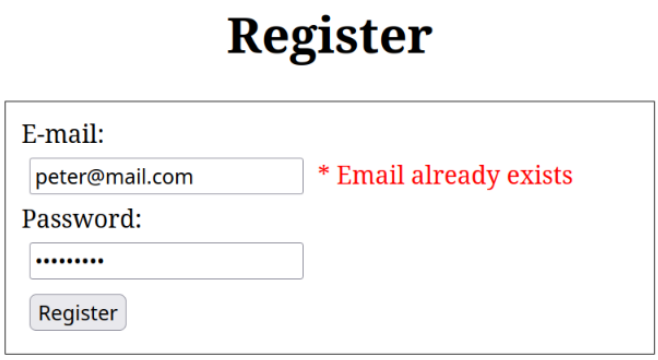

Figure 2: Existent email

To do this check, we add the needed code to the form after the basic validation:

```
...
$errors = $login->validate();

if (empty($errors) && LoginRepository::searchByEmail($login) > 0) {
    $errors['email'] = "* Email already exists";
}
...
```

We also need to create a new static method in the `LoginRepository` class for searching an existent email, the `searchByEmail` method:

```
public static function searchByEmail($object): int {
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        $stmt = $conn->prepare("SELECT email FROM login WHERE
        ↪  email=:email");
        $stmt->execute(['email'=>$object->getEmail()]);
        return $stmt->rowCount();
    }
    return 0;
}
```

Once all the validation is passed, we add the new login and password to the database, but before, we need to hash the password using the functions of the Unit 6 point 7:

```
if (empty($errors)) {
    if(isset($_POST['submit']) ){
        //New login
        try {
            $login->setPassword(password_hash($login->getPassword(),
            ↪  PASSWORD_DEFAULT));
            LoginRepository::insert($login);
            $opMsg = "Register successful";//If no errors, make a new empty
            ↪  login to clear the fields
            $login = new Login();
        } catch (Exception $e) {
            echo "Error registering user: " . $e->getMessage();
        }
```

```
    }
}
```

If you query the database, you can see the new login hashed:

```
select * from login;
+----+----------------+--------------------------------------------
----------------+
| id | email          | password
+----+----------------+--------------------------------------------
----------------+
|  1 | admin@mail.com | $2y$10$ET9Sj8II1XDQpIYBN4RQ9.lQ9IsdYhrTMcUpOwkXcCaMVEudd
+----+----------------+--------------------------------------------
----------------+
```

2.4 Login

Once a user is registered, it need a login form to be authenticated. We will create the `login_form.php` script to do that:

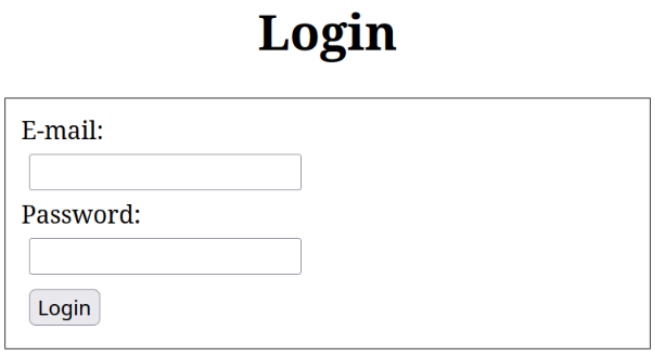

Figure 3: Login form

In this form, once the basic validation is done, we need to check if both the email and password exist and are correct. If they don't match, we show only an error message for security reasons. First of all, let's create a new method in `LoginRepository` to check if the stored credentials are valid, using the `password_verify` method seen on Unit 6:

```php
public static function checkCredential($clearPassword, $object): bool|int {
    $conn = DBConnection::connectDB();
    if (!is_null($conn)) {
        $stmt = $conn->prepare("SELECT * FROM logins WHERE email=:email");
        $stmt->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE,
        ↪  'Login');
        $stmt->execute(['email'=>$object->getEmail()]);
        $login = $stmt->fetch();
        if($login && password_verify($clearPassword,
        ↪  $login->getPassword())){
            //Returns the id login if successful
            return $login->getId();
        } else {
            return false;
        }
    }
    return false;
}
```

Then, in `login_form.php`, use the new method to check if the credentials match those stored in the DB. If they are, store the user's id in a session variable in order to know in other pages if the user is authenticated (remember to call `session_start()` at the beginning of the script) and redirect to the `provider_list.php` script:

```
if (empty($errors)) {
    if(isset($_POST['submit']) ){
        try {
            $idLogin = LoginRepository::checkCredential($clearPassword,
            ↪  $login);
            if($idLogin){
                $opMsg = "Login successful";
                $_SESSION['user_id'] = $idLogin;
                header('Location: provider_list.php');
            } else {
                $opMsg = "Login failed";
            }
            $login = new Login();
        } catch (Exception $e) {
            echo "Error checking login: " . $e->getMessage();
        }
    }
}
```

2.5 Tracking user and logging out

Once the user is authenticated, we can check if it is logged in on each script using the session variable `'user_id'`. Make a new partial, named `nav_bar.part.php` to be included at the top of each page with that shows the user's email:

```
<?php
require_once __DIR__ . '/../models/Login.php';
require_once __DIR__ . '/../models/LoginDao.php';

$welcomeMessage = '';
if(isset($_SESSION['user_id'])) {
    $user = LoginDao::select($_SESSION['user_id']);
    if (!is_null($user)) {
        $userEmail = LoginDao::select($_SESSION['user_id'])->getEmail() ??
        ↪  '';
        $welcomeMessage = "Welcome $userEmail";
```

```
    }
}
?>
<!-- nav_bar.part.php -->
<nav>
    <span> <?= $welcomeMessage ?> </span>
</nav>
```

Then, include it in `login_form.php` and in `login_list.php` (remember to call `session_start()` at the beginning of the script):

```
// login_form.php
<body>
<?php include __DIR__ . '/partials/nav_bar.part.php'; ?>
<h1>Edit login</h1>
```

In `login_form.php`, we can get the login data using the `id` and, then, pass it to the form:

```
//Read id from login_list
if($_SERVER['REQUEST_METHOD'] == "GET") {
    if (!empty($_REQUEST["id"])) {
        $login = LoginDao::select($_REQUEST['id']);
    } else {
        //Read id from session
        if(isset($_SESSION['user_id'])){
            $login = LoginDao::select($_SESSION['user_id']);
        }
    }
}
```

The next step is to implement a method to logging out the user. In `nav_bar.part.php` add a link only if the user is authenticated:

```
<!-- nav_bar.part.php -->
<nav>
    <span> <?= $welcomeMessage ?> </span> |
    <span>
        <?php
        if(isset($_SESSION['user_id'])) {
```

```
            echo "<a href='logout.php'>Logout</a>";
        }
        ?>
    </span>
</nav>
```

The target, `logout.php`, simply deletes the session variable and redirect to the login access form:

```
<?php
session_start();

if(isset($_SESSION['user_id'])) {
    unset($_SESSION['user_id']);
    header('Location: login_access_form.php');
}
```

2.6 Permissions

To establish an authorization system, we need each user to having a role. Each role will determine which pages and actions can the user do.

For our application, we are going to establish 3 different roles:

- Unauthenticated users: can only visit the login and register pages.
- Users with the role **'user'**: can edit their own user data in the `login_form.php` script, but cannot access to the `logins_list` page.
- Users with the role **'admin'**: can acces to the `login_list.php` page and edit all the users data in `login_form.php`.

Before coding, we need another field in our logins table to store the role. You can create it with phpMyAdmin or executing this SQL statement:

```
ALTER TABLE `logins` ADD `role` VARCHAR(20) NULL DEFAULT 'user';
```

The default role will be 'user'. We are allowing null data, so the existing users will be equivalent to users with the role 'user'.

Go to the register form and create a new user for the admin role. Inspect your database and verify that has the role `'user'`. Change it to `'admin'`:

Figure 4: admin user in phpMyAdmin

Make sure you have another user with the role 'user'. Now we can change the permissions in each page.

In addition, we need a new property in our `Login` class with the name `'role'` and their setter and getter:

```php
private string $role;

public function getRole(): string
{
    return $this->role;
}

public function setRole(string $role): void
{
    $this->role = $role;
}
```

Also, add the necessary code to the `LoginRepository` class to store and retrieve the new field.

In `logins_list.php`, simply add a check in the beginning of the script. If the user is authenticated but doesn't has the admin role, redirect it to `login_form.php`. If the user is not authenticated, redirect to `login_access_form.php`:

```php
if(isset($_SESSION['user_id'])) {
    $user = LoginDao::select($_SESSION['user_id']);
    $userRole = $user->getRole();
    if($userRole != 'admin'){
        //Has the role user
        header('Location: login_form.php');
    }
} else {
    //No authenticated user
    header('Location: login_access_form.php');
}
```

Go to the `login_form.php`. In this form the admin users can view and change all the users data, but the normal users only can view and change their own data. To do that, simply add a new condition to the code to check the user's id (remember that a user can change the query string in the browser):

```php
if(isset($_SESSION['user_id'])) {
    $user = LoginRepository::select($_SESSION['user_id']);
    $userRole = $user->getRole();
    if($userRole != 'admin' && $login->getId() != $user->getId()){
        //Has the role user and the same id
        header('Location: provider_list.php');
    }
} else {
    //No authenticated user
    header('Location: login_form.php');
}
```

There are a lot of things to improve, like adding the possibility to change the user's role or a mechanism to change the user's password correctly. You can do it as exercise.

Get the full app code in the GitHub repo.